

# Fusing Loopless Algorithms for Combinatorial Generation

Stephen Scott Violich  
(*B.Sc., M.E.M., University of Canterbury*)

A Thesis submitted in partial fulfilment of the requirements  
for the Degree of Master of Science in Computer Science

Examiners: Professor Tadao Takaoka (University of Canterbury)  
Professor Mike Atkinson (University of Otago)

University of Canterbury  
2006



To Petra, with whom I have lived, learned and loved most.



## Abstract

Loopless algorithms are an interesting challenge in the field of combinatorial generation. These algorithms must generate each combinatorial object from its predecessor in no more than a constant number of instructions, thus achieving theoretically minimal time complexity. This constraint rules out powerful programming techniques such as iteration and recursion, which makes loopless algorithms harder to develop and less intuitive than other algorithms.

This thesis discusses a divide-and-conquer approach by which loopless algorithms can be developed more easily and intuitively: fusing loopless algorithms. If a combinatorial generation problem can be divided into subproblems, it may be possible to conquer it looplessly by fusing loopless algorithms for its subproblems. A key advantage of this approach is that it allows existing loopless algorithms to be reused. This approach is not novel, but it has not been generalised before.

This thesis presents a general framework for fusing loopless algorithms, and discusses its implications. It then applies this approach to two combinatorial generation problems and presents two new loopless algorithms. The first new algorithm, MIXPAR, looplessly generates well-formed parenthesis strings comprising two types of parentheses. It is the first loopless algorithm for generating these objects. The second new algorithm, MULTPERM, generates multiset permutations in linear space using only arrays, a benchmark recently set by Korsh & LaFollette (2004). Algorithm MULTPERM is evaluated against Korsh & LaFollette's algorithm, and shown to be simpler and more efficient in both space and time.



## Table of Contents

<b>Chapter 1:</b>	<b>Introduction</b>	<b>1</b>
1.1	Combinatorial Generation . . . . .	3
1.2	Loopless Algorithms . . . . .	5
1.3	Fusing Loopless Algorithms . . . . .	9
1.4	Research Overview . . . . .	9
<b>Chapter 2:</b>	<b>Literature Review</b>	<b>11</b>
2.1	Ehrlich’s Loopless Algorithms . . . . .	11
2.2	Williamson’s Loopless Algorithm . . . . .	12
2.3	Fused Loopless Algorithms for Multiset Permutations . . . . .	13
<b>Chapter 3:</b>	<b>Fusing Loopless Algorithms</b>	<b>17</b>
3.1	Fusing by Nesting . . . . .	17
3.2	Reinitialising . . . . .	18
3.3	Fusing with Williamson’s Algorithm . . . . .	20
<b>Chapter 4:</b>	<b>Generating Mixed Parenthesis Strings</b>	<b>22</b>
4.1	Mixed Parenthesis Strings . . . . .	22
4.2	Finding Right Parentheses . . . . .	26
4.3	Xiang & Ushijima’s Algorithm . . . . .	28
4.4	Algorithm MIXPAR . . . . .	32
<b>Chapter 5:</b>	<b>Generating Multiset Permutations</b>	<b>38</b>
5.1	Multiset Permutations . . . . .	38
5.2	Chase’s Algorithm . . . . .	41
5.3	Algorithm MULTPERM . . . . .	46
5.4	Evaluation vs. Korsh & LaFollette’s Algorithm . . . . .	51
<b>Chapter 6:</b>	<b>Discussion</b>	<b>54</b>
6.1	Results . . . . .	54

6.2 Future Work . . . . .	55
<b>References</b>	<b>57</b>
<b>Appendix A: grayrec.c</b>	<b>60</b>
<b>Appendix B: graylpl.c</b>	<b>62</b>
<b>Appendix C: will.c</b>	<b>64</b>
<b>Appendix D: xiang.c</b>	<b>66</b>
<b>Appendix E: mixpar.c</b>	<b>68</b>
<b>Appendix F: chase.c</b>	<b>71</b>
<b>Appendix G: chasemod.c</b>	<b>74</b>
<b>Appendix H: multperm.c</b>	<b>77</b>
<b>Appendix I: korsh.c</b>	<b>81</b>
<b>Appendix J: timer.py</b>	<b>89</b>
<b>Appendix K: timedata.txt</b>	<b>91</b>



## Acknowledgments

I would like to thank a number of people for their contributions towards this thesis: my Senior Supervisor, Professor Tadao Takaoka, for guiding me to achieve all of my goals for this research; my Co-Supervisor, Associate Professor Tim Bell, for invaluable advice throughout this degree; my predecessor, Dr. Shane Saunders, for showing how a thesis on algorithms should be written; my colleagues, Michael JasonSmith and Oliver Hunt, for technical support and intellectual diversion; my academic big brothers, Haydon Cherry and Ken Courtney, for encouraging and inspiring me; my great friend, Tristan Shepherd, for reconnecting me with my greatest passion; and finally Petra Gimbad, simply for being Petra Gimbad.



# Chapter I

## Introduction

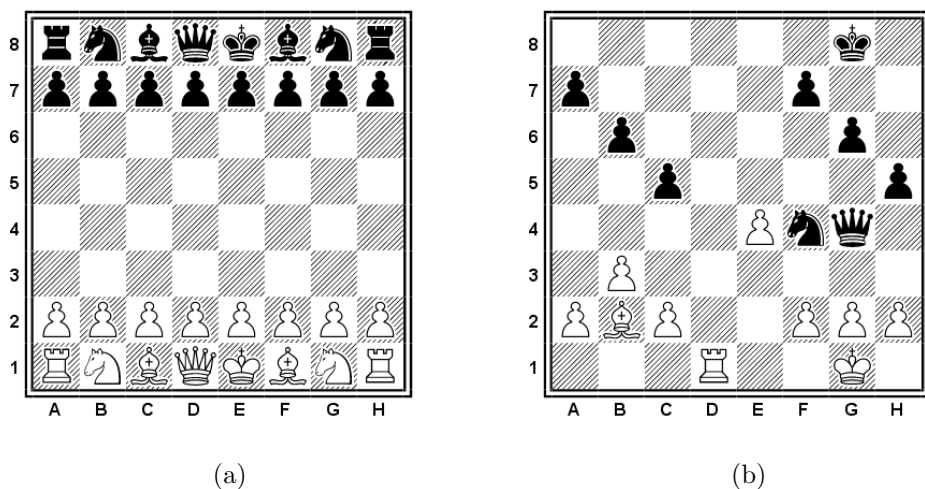
“Chess players are madmen of a certain quality,  
the way the artist is supposed to be, and isn't, in general.”

Marcel Duchamp (1887-1968)



A chess master and influential artist, Duchamp was well qualified to comment. He has since been contradicted, however, by the rise of a new type of chess player that is neither mad nor artistic: the chess-playing computer. Computer chess came of age when IBM's *Deep Blue* beat Garry Kasparov, the reigning human world champion, under tournament conditions in 1997. Given that computer technology continues to improve, it seems inevitable that one day all of the world's best chess players will be computers. How do machines succeed in a game that supposedly favours artistic madness?

Chess is a game of many, though finite, possible positions. The number-crunching power of computers is such that they can generate and evaluate many more of these possible future positions than humans can. A good human player considers perhaps a few dozen positions, three to five turns in



**Figure 1.1** (a) The starting configuration of a chessboard, from which each side has 20 possible moves. (b) A composed chess problem, *white to move and mate in 2*.

advance, relying on intuition to prune out all but the most promising. *Deep Blue*, on the other hand, could generate *all* possible positions *six* turns in advance.

To understand just how many positions this involves generating, consider that the standard chess starting position shown in Figure 1.1(a) offers white 20 possible moves and black 20 possible replies. Therefore, a computer player looking one turn ahead must generate  $20 \times 20 = 400$  possible positions resulting from both sides' opening moves. Conservatively assuming that each side continues to have 20 possible moves per turn, a computer player looking three turns ahead generates 64 million positions, while *Deep Blue* looking six turns ahead generated over 4 *quadrillion* ( $4 \times 10^{15}$ ). More recent systems such as *Hydra* have raised the bar to a staggering nine turns.

Raw processing power, then, is a more than adequate substitute for madness. As Kasparov said: "What I do by just feeling that it's right or wrong ... [the] machine finds by just making these billions and billions of calculations."

Generating all those chess positions is a real-world example of combinatorial generation, the subject area of this thesis. In particular, this thesis deals with loopless algorithms for combinatorial generation, which can be

likened to a chess variant: composed chess problems. A chess problem comprises a starting position and an objective, an example of which is shown in Figure 1.1(b), and demands optimal use of the limited pieces and moves. Similarly, loopless algorithms demand efficient use of limited programming structures and time.

This chapter introduces the fields of combinatorial generation and loopless algorithms, and gives examples of each by looking at two algorithms, one recursive and one loopless, for a simple generation problem. It also introduces the specific topic of this thesis, the approach of developing new loopless algorithms by fusing existing ones. The chapter concludes with an overview of the rest of the thesis.

### 1.1 *Combinatorial Generation*

Combinatorial generation is the branch of computer science that studies algorithms for generating combinatorial objects, which are sets of objects that satisfy specified criteria. Examples of combinatorial objects include combinations, permutations and well-formed parenthesis strings, and the chess positions discussed in the introductory section. This area is well studied, covered by authors such as Nijenhuis & Wilf (1975), Reingold, Nievergelt & Deo (1977), Wilf (1989) and Savage (1997).

A combinatorial generation algorithm has the task of generating all combinatorial objects in a given set, for example generating all binary strings of length  $n$  as shown in Figure 1.2. Such algorithms should be designed to use time and space efficiently, and often to generate the objects according to some order or property. Two properties relevant to this thesis are the

1. 0 0 0 0	5. 0 1 1 0	9. 1 1 0 0	13. 1 0 1 0
2. 0 0 0 1	6. 0 1 1 1	10. 1 1 0 1	14. 1 0 1 1
3. 0 0 1 1	7. 0 1 0 1	11. 1 1 1 1	15. 1 0 0 1
4. 0 0 1 0	8. 0 1 0 0	12. 1 1 1 0	16. 1 0 0 0

**Figure 1.2** Binary strings of length  $n = 4$  in Gray code order, as generated by algorithms GRAYREC and GRAYLPL.

**Algorithm 1** GRAYREC, a recursive algorithm for generating binary strings of size  $n$  in Gray code order.

```

    /* Main. */
1.  read  $n$ 
2.  for  $i = 1$  to  $n$  do  $bin[i] = 0$ 
3.  print  $bin$ 
4.   $next(1)$ 

    /* Recursively generates all binary strings rooted at  $bin[j]$ . */
5.  procedure  $next(j)$  {
6.      if  $j$  is  $n$  then {
7.          invert  $bin[j]$ 
8.          print  $bin$ 
9.      } else {
10.          $next(j + 1)$ 
11.         invert  $bin[j]$ 
12.         print  $bin$ 
13.          $next(j + 1)$ 
14.     }
15. }
```

Minimal Change Property (MCP) and the Strong Minimal Change Property (SMCP). Both the MCP and the SMCP imply that each object in a listing differs from its predecessor by only one element; the SMCP also requires that the lone differing elements be at the same position within the object. Listings with these properties are often efficient to generate, since there are only  $O(1)$  changes between successive objects. As will be discussed in the next section, this is all loopless algorithms can afford.

GRAYREC, shown in Algorithm 1, is a combinatorial generation algorithm for generating all binary strings of length  $n$ . It generates the strings in Gray (1953) code order, as shown in Figure 1.2, meaning there is exactly one bit change between each successive string. Thus, GRAYREC's listings have the SMCP, which makes for a simple and efficient algorithm. A complete C program for GRAYREC is given in Appendix A.

Referring to Algorithm 1, GRAYREC works as follows. The first step is to initialise all bits to 0 (line 2). Next, all strings are generated by calling the recursive procedure for position 1 (line 4). The recursive step generates all strings rooted at some position  $j$  by generating all strings rooted at  $j + 1$ , inverting bit  $j$ , then generating all strings rooted at  $j + 1$  again (lines 10–13). The non-recursive step occurs at position  $n$ , where all strings rooted at  $n$  are generated simply by inverting bit  $n$  (lines 7–8). For comparison, a loopless algorithm for the binary strings problem will be examined in the next section.

## 1.2 Loopless Algorithms

Loopless algorithms, introduced by Ehrlich (1973), present an interesting challenge in the field of combinatorial generation. These algorithms must generate each combinatorial object from its predecessor in no more than a constant number of instructions. Thus, each object is generated in  $O(1)$  time. This means that powerful programming structures such as recursion and looping cannot be used in the code for generating each successive object (although one loop is required to generate *all* objects).

GRAYLPL, shown in Algorithm 2, is a loopless algorithm for generating all binary strings of length  $n$  in Gray code order. It generates exactly the same output as GRAYREC (Figure 1.2), but achieves this using only  $O(1)$  instructions per object. It is not a radically different algorithm from GRAYREC; rather it looplessly simulates the recursive process. The technique it uses to achieve this reappears in all but one of the subsequent loopless algorithms in this thesis. This technique will be discussed in further detail after the general structure of loopless algorithms. A complete C program for algorithm GRAYLPL is given in Appendix B.

The main section of GRAYLPL has a structure typical of loopless algorithms in general. First, the algorithm is initialised (lines 2–4), which includes establishing the first object. Then, a loop iteratively calls a procedure that generates the next object in  $O(1)$  time (line 7). The loop terminates when some condition equivalent to whether the last object has been generated evaluates to true (line 6). Initialisation is permitted  $O(n)$  time, where  $n$  is the number of bits in the binary object. This is the minimum complexity pos-

**Algorithm 2** GRAYLPL, a loopless algorithm for generating binary strings of size  $n$  in Gray code order.

```

    /* Main. */
1.  read  $n$ 
2.  for  $i = 1$  to  $n$  do  $bin[i] = 0$ 
3.  for  $i = 0$  to  $n$  do  $e[i] = i$ 
4.   $j = n$ 
5.  print  $bin$ 
6.  while  $j$  is not 0 do {
7.      next
8.      print  $bin$ 
9.  }

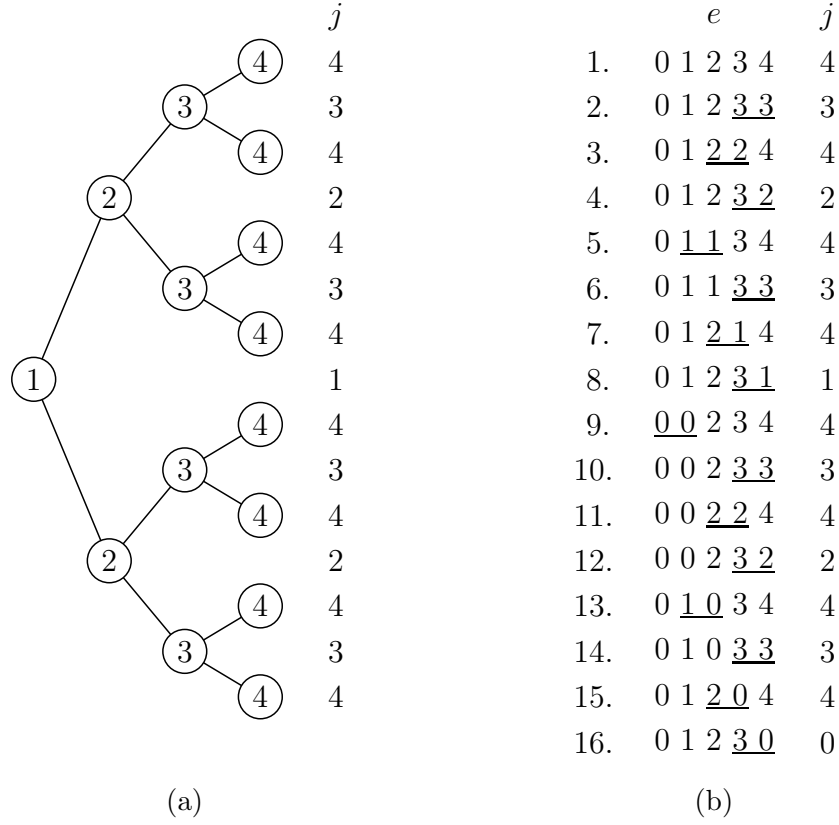
    /* Looplessly generates the next binary string. */
10. procedure next {
11.      $e[n] = n$ 
12.     invert  $bin[j]$ 
13.      $e[j] = e[j - 1]$ 
14.      $e[j - 1] = j - 1$ 
15.      $j = e[n]$ 
16. }

```

sible, since each of the  $n$  bits must be initialised. The generation procedure and termination condition must both run in  $O(1)$  time, which are also minimum possible complexities. Pedantically speaking, only the procedure *next* is loop-free, but Ehrlich's definition of looplessness encompasses the whole algorithm. Input and output statements are typically ignored during discussion, since they are necessary trivialities not directly related to generation. In other words, the algorithm generates the objects regardless of whether they are printed to standard output or used in some other way.

The technique for looplessly simulating recursion uses array  $e$  as a sort of conveyor belt, with variable  $j$  positioned after  $e[n]$ . Variable  $j$  is the position of the bit to invert during an iteration of procedure *next*. For example, if  $j = 4$  at the beginning of an iteration, then the 4th bit will be inverted during





**Figure 1.3** (a) GRAYREC uses an inorder traversal of a balanced binary tree of height  $n = 4$  to select values for  $j$ . (b) GRAYLPL looplessly simulates this process using array  $e$ , in which  $e[j]$  and  $e[j - 1]$  have been underlined at each step for clarity.

that iteration. Information moves along array  $e$  by the following mechanism: when some  $e[j]$  gets the value from  $e[j - 1]$ ,  $e[j - 1]$  is reset to its initial value of  $j - 1$  (lines 13–14); similarly, when  $j$  gets the value from  $e[n]$ ,  $e[n]$  is reset to its initial value of  $n$  at the start of the next iteration (lines 15 and then 11).

As can be seen from Figure 1.3, this mechanism iteratively selects bit  $j$  for inversion in exactly the same order as the inorder tree traversal of GRAYREC. Resetting  $e[n]$  to  $n$  at the start of each iteration ensures that, if  $j \neq n$  in one iteration, then  $j = n$  for the next; this has the effect of falling

from a branch node to a leaf node. The operations involving  $e[j]$  and  $e[j-1]$  ensure that once a value has been selected for  $j$ , it will be skipped in favour of a lesser value before being selected again; this has the effect of rising from a leaf node to the correct branch node.

Loopless algorithms have two main practical disadvantages. An informal empirical evaluation of GRAYREC and GRAYLPL suggested that the recursive algorithm ran in about two-thirds of the time taken by the loopless algorithm (for sufficiently large  $n \geq 26$ ). This confirms the conventional view that loopless algorithms, which run in  $O(1)$  worst-case time per object, run slower than their recursive counterparts, which run in  $O(n)$  worst-case time per object (though  $O(1)$  amortised time per object). Thus, the extra book-keeping required to achieve looplessness is a significant cost. The other main disadvantage is that, since they must be developed without using recursion or loops, loopless algorithms are less intuitive and more difficult to design than recursive algorithms.

In the absence of any practical advantage, why study loopless algorithms? Donald Knuth suggests that it is both an interesting and instructive task, commenting:

“The extra contortions that we need to go through in order to achieve looplessness are usually ill-advised, because they actually cause the total execution time to be longer than it would be with a more straightforward algorithm. But hey, looplessness carries an academic cachet. So we might as well treat this task as a challenging exercise that might help us to sharpen our algorithmic wits.”<sup>1</sup>

and

“Loopless algorithms tend to run slower than their loopy counterparts, especially in cases like the present where the additional overhead needed to avoid looping appears to be substantial. So the search for a loopless implementation is strictly academic. Yet it still was fascinating enough to keep me working on it for three days during my recent vacation.”<sup>2</sup>

---

<sup>1</sup> <http://www-cs-faculty.stanford.edu/~knuth/programs/spiders.w>

<sup>2</sup> <http://www-cs-faculty.stanford.edu/~knuth/programs/spspan.w>

Thus, it is the added difficulty, and perhaps even style, of loopless algorithms that ensures their survival as an academic pursuit, much like the composed chess problems mentioned in the introductory section. Just as chess problems might have to be solved without powerful pieces like queens or rooks, loopless algorithms require the programmer to solve problems without the usual powerful structures like loops or recursion. Instead of *mate in 2*, the objective is *generate in  $O(1)$* .

### 1.3 Fusing Loopless Algorithms

The specific topic of this thesis is that of fusing loopless algorithms. Fusing refers to the joining or combining of two loopless algorithms in such a way that the resulting algorithm is also loopless. In theory, fusing the constant-time parts of two loopless algorithms produces code that still runs in constant time, since  $O(1) + O(1) = O(1)$ . This means that a divide-and-conquer approach may be taken to the development of loopless algorithms: if a combinatorial generation problem can be divided into subproblems, it may be possible to conquer it looplessly by fusing loopless algorithms for its subproblems.

This approach would appear to have several benefits. First, the subproblems would be easier to solve than the original problem. Second, it may be possible to reuse existing algorithms to solve the subproblems. Finally, the resulting algorithm would likely be more intuitive, since it would be modular rather than monolithic. In practice, however, this approach has been used relatively few times in the literature, some examples of which are covered in Chapter 2. Furthermore, it has only been used as a means to an end, and not discussed as a general technique.

### 1.4 Research Overview

This thesis addresses two specific questions. First, can the approach of fusing loopless algorithms be generalised? Second, is the approach a generally useful tool for researchers of loopless algorithms? To answer these questions, this thesis presents a general framework for fusing loopless algorithms, applies the framework to develop two new loopless algorithms, and discusses the results.

This research is arranged as follows. Relevant algorithms from the literature are covered in Chapter 2. These include Williamson’s algorithm, a seminal loopless algorithm from which many are derived, and prior examples of the fusion approach. A general framework is developed and its implications discussed in Chapter 3. The framework is applied to develop a loopless algorithm for generating mixed parenthesis strings in Chapter 4, and a loopless algorithm for generating multiset permutations in Chapter 5. The multiset permutations algorithm is experimentally evaluated against one recently published by Korsh & LaFollette (2004). Conclusions are drawn and future directions indicated in Chapter 6.

A paper by Takaoka & Violich (2006) presenting the material covered in Chapters 3–5 was accepted for publication at a refereed conference, the twelfth *Computing: The Australasian Theory Symposium*.

## Chapter II

### Literature Review

This chapter covers the literature upon which this thesis builds, beginning with Ehrlich's introduction and definition of loopless algorithms. It then discusses Williamson's algorithm, a seminal loopless algorithm upon which many of the subsequent algorithms in this thesis are based. Finally, it covers prior examples from the literature of loopless algorithms developed by a fusion approach.

#### **2.1 Ehrlich's Loopless Algorithms**

Ehrlich introduced the idea of loopless algorithms for combinatorial generation. Prior algorithms were typically coded as a procedure that produced a new combinatorial object each time it was called. All known examples of such procedures contained at least one loop that required  $O(n)$  iterations. Ehrlich described a method by which procedures could generate combinatorial objects and yet not contain such loops, hence the name. He hinted that this approach might reduce the mean time required to generate each object, but it has been borne out that loopless algorithms are generally slower than their counterparts, as mentioned in Section 1.2.

Ehrlich's definition of a loopless algorithm consists of the following four criteria, where  $n$  is the number of elements in the objects being generated:

1. Generating the first object takes  $O(n)$  time.
2. Generating each successive object takes  $O(1)$  time.
3. Testing for the last object takes  $O(1)$  time.
4. The objects are represented in a simple form and can be read directly.

The first three requirements all represent the theoretical minimum time complexities that those tasks could possibly require. The fourth point implies that the algorithm completely generates the actual objects, rather than perhaps saving time by generating some other notations that in turn require  $O(n)$  translation. As will be discussed later, this fourth point is usually interpreted to mean that the objects must be stored in arrays. The use of linked lists by Korsh & Lipschutz (1997) is a point of contention, since elements in a linked list cannot be read directly. This is because accessing any element in a linked list takes  $O(n)$  time, compared to  $O(1)$  time for an array. It can be argued, however, that accessing the entire object using either structure takes  $O(n)$  time.

Ehrlich gave a general algorithm for generating set partitions, and demonstrated modifications of it for permutations, combinations, and other set partitions. His general algorithm includes two important concepts common to most of the algorithms in this thesis. The first is that of looplessly simulating element selection by inorder tree traversal, although Ehrlich's version was not as simple or efficient as Williamson's, covered in Section 1.2 and discussed again in the next section. The other important concept in Ehrlich's general algorithm was that of elements moving along paths between their minimum and maximum values in certain directions, maintained in array  $d$ . This technique can also be seen in Williamson's algorithm in the next section.

## **2.2 Williamson's Loopless Algorithm**

Williamson's loopless algorithm for generating variations in Gray code order, shown in Algorithm 3 on 14, is a basis for many of the subsequent algorithms in this thesis. It generates elements of the product space  $S = S_1 \times S_2 \times \dots \times S_n$  where  $S_i = \{0, 1, \dots, r_i - 1\}$ , as shown in Figure 2.1. In other words, the first element varies through the values in  $S_1$ , for each of which the second element varies through the values in  $S_2$ , and so on. A complete C program for Williamson's algorithm is given in Appendix C.

Referring to Algorithm 3, Williamson's algorithm works as follows. The current variation is stored in array *var*, initialised such that all elements are

1. 0 0 0	7. 1 1 0	13. 2 0 0
2. 0 0 1	8. 1 1 1	14. 2 0 1
3. 0 0 2	9. 1 1 2	15. 2 0 2
4. 0 1 2	10. 1 0 2	16. 2 1 2
5. 0 1 1	11. 1 0 1	17. 2 1 1
6. 0 1 0	12. 1 0 0	18. 2 1 0

**Figure 2.1** Variations in Gray code order generated by Williamson’s algorithm for input  $r = \{3, 2, 3\}$ .

minimal (line 3). Each call to procedure *next* increments or decrements the selected  $j$ th element according to direction  $d[j]$  (line 14). As mentioned in Section 2.1, the idea of moving elements along paths in particular directions was introduced by Ehrlich. Following the change, if element  $var[j]$  becomes extremal (line 15), then its direction is reversed (line 18). Throughout this process, the selection of the  $j$ th element is performed using array  $e$  (lines 13, 16–17, and 20). As covered in Section 1.2, this technique mimics the selection pattern of a recursive algorithm, by looplessly simulating the inorder traversal of a balanced binary tree of height  $n$ .

Williamson’s algorithm is a general loopless algorithm, in that it can be modified to generate many different types of combinatorial object in Gray code order. Xiang & Ushijima (2001), for example, derive loopless algorithms for generating combinations and well-formed parenthesis strings respectively from Williamson’s algorithm. Notably, Williamson’s algorithm is a common basis for loopless algorithms for multiset permutations, several examples of which will be covered in the next section. The approach is to fuse a loopless algorithm for generating combinations within Williamson’s algorithm. This kind of fusion is covered in Section 3.3.

### 2.3 Fused Loopless Algorithms for Multiset Permutations

A multiset is a set with repetitions, for example  $\{1, 1, 1, 2, 2, 3\}$ . A common approach in the design of loopless algorithms for generating multiset permutations is to follow the well-known process for generating set permutations,

**Algorithm 3** Williamson’s loopless algorithm for variations in Gray code order.

```

    /* Main. */
1.  read  $n$ 
2.  for  $i = 1$  to  $n$  do read  $r[i]$ 
3.  for  $i = 1$  to  $n$  do  $var[i] = 0$ 
4.  for  $i = 0$  to  $n$  do  $e[i] = i$ 
5.  for  $i = 1$  to  $n$  do  $d[i] = 1$ 
6.   $j = n$ 
7.  print  $par$ 
8.  while  $j$  is not 0 do {
9.       $next$ 
10.     print  $var$ 
11. }

    /* Looplessly generates the next variation. */
12. procedure  $next$  {
13.      $e[n] = n$ 
14.     add  $d[j]$  to  $var[j]$ 
15.     if  $var[j]$  is either 0 or  $r[j] - 1$  then {
16.          $e[j] = e[j - 1]$ 
17.          $e[j - 1] = j - 1$ 
18.          $d[j] = -d[j]$ 
19.     }
20.      $j = e[n]$ 
21. }
```

but adapt it for moving groups of elements of the same value, for example the 1s. This is illustrated in Figure 2.2, which shows set permutations generated by Johnson’s (1963) and Trotter’s (1962) algorithm, and multiset permutations generated by Korsh & LaFollette’s algorithm.

Set permutations are generated by moving a selected element  $i$  through all positions in the subpermutation that comprises all elements  $\leq i$ . Thus, in Figure 2.2(a), the 3 moves through the two remaining positions in the entire



1. 1 2 3	1. 1 2 3 3	7. 3 3 2 1
2. 1 3 2	2. 1 3 3 2	8. 3 2 3 1
3. 3 1 2	3. 1 3 2 3	9. 3 2 1 3
4. 3 2 1	4. 3 1 2 3	10. 2 3 1 3
5. 2 3 1	5. 3 1 3 2	11. 2 3 3 1
6. 2 1 3	6. 3 3 1 2	12. 2 1 3 3
(a)	(b)	

**Figure 2.2** (a) Set permutations of generated by the Johnson-Trotter algorithm. (b) Multiset permutations generated by Korsh and LaFollette's algorithm.

permutation (lines 1–3). When the 3 has finished, the 2 is moved once (line 4), which in this example completes its travels through the subpermutation consisting of itself and the 1. Finally, the 3 moves back through the possible positions to complete the listing (lines 5–6).

Multiset permutations are generated by much the same process, except that similar elements must be moved through all possible positions as a group. Thus, in Figure 2.2(b), the 3s move through the six possible configurations for two elements in four spaces (lines 1–6). Again, the 2 is moved once (line 7), and then the 3s reverse their journey (8–12).

This grouped element movement can be achieved using a combinations algorithm. Fusing a loopless algorithm for combinations with Williamson's algorithm for element selection has been a common approach for developing algorithms for multiset permutations. This section now briefly covers three such algorithms based on this approach, another two based on differing approaches, and hints at how the approach is used in Chapter 5 to develop a new algorithm.

The first loopless algorithm developed using this approach was that of Korsh & Lipschutz. It was the first loopless algorithm to generate multiset permutations in constant time, and also achieved only  $O(n)$  space. The combinations algorithm used for grouped element movement was that of Lehmer (1964). Despite generating each permutation in  $O(1)$  time, the algorithm

seems to lie outside Ehrlich's definition of a loopless algorithm. This is because Korsh & Lipschutz's algorithm uses linked lists to represent the permutations, arguably conflicting with Ehrlich's fourth point, that objects must be represented in a simple form. This seems to preclude linked lists, since accessing any single element in the permutation requires  $O(n)$  operations.

The second such algorithm was developed by Vajnovszki (2003), by fusing Eades & McKay's (1984) algorithm for combinations with Williamson's algorithm. He applied a mathematical technique known as shuffle on trajectories to combinatorial generation. The algorithm uses only arrays for storage, and requires  $O(kn)$  space, where  $k$  is the number of distinct elements and  $n$  is the total number of elements.

The third algorithm was that recently published by Korsh & LaFollette, produced by fusing the same two component algorithms as Vajnovszki. Korsh & LaFollette's was the first loopless algorithm for multiset permutations to achieve  $O(n)$  space complexity. Since this algorithm relies only on arrays for storage, it satisfies Ehrlich's definition of a loopless algorithm, avoiding the controversy encountered by Korsh & Lipschutz's use of linked lists.

Loopless algorithms for multiset permutations taking a different approach include those by Canfield & Williamson (1995) and Takaoka (1999). Canfield & Williamson's is a loopless version of Pruesse & Ruskey's (1994) algorithm for generating linear extensions of a poset. Takaoka's algorithm uses the technique for loopless element selection covered in Section 1.2, but differs by using a novel tree traversal approach for transposing elements.

The algorithm presented in Chapter 5 is also developed by the approach of fusing a combinations algorithm with Williamson's algorithm. It was hoped that it would be the first loopless,  $O(n)$ -space algorithm for multiset permutations, until the publication of Korsh & LaFollette (the two algorithms were developed independently). The new algorithm and that of Korsh & LaFollette are empirically compared in Section 5.4, the former proving to be simpler and more efficient in time and space.

## Chapter III

### Fusing Loopless Algorithms

This chapter develops a general framework for fusing loopless algorithms. As mentioned in Section 1.3, loopless algorithms can be fused to produce new loopless algorithms. This chapter looks at two structures for fusing: nesting, and modifying Williamson’s algorithm. Both structures fuse one loopless algorithm within another, meaning that the inner algorithm must generate many complete listings in succession, which loopless algorithms are generally not designed to do. Thus, this chapter also covers issues related to *reinitialising* loopless algorithms, so that they can be run repeatedly while maintaining  $O(1)$  time per object.

#### **3.1 Fusing by Nesting**

The general structure of loopless algorithms was introduced discussing GRAYREC in Section 1.2. First, the algorithm is initialised, and the first object generated, in  $O(n)$  time. Then, each successive object is generated in  $O(1)$  time, using a loop. The loop terminates when the last object has been generated, the condition for which must be testable in  $O(1)$  time. This structure is shown in Figure 3.1(a).

The general structure can be extended to nest one loopless algorithm within another. Figure 3.1(b) shows a general structure for nesting, in which procedures and functions belonging to the inner algorithm have the suffix 2, and those for the outer algorithm have the suffix 1. Following initialisation (line 1), for each iteration of the outer algorithm (line 7), the inner algorithm completes a full cycle (lines 3–5). After each full cycle, the inner algorithm must be reinitialised, or prepared to run again. This structure allows all combinations of states of both algorithms to occur.

```

1.  init
2.  do {
3.      next
4.  } while not last

```

(a)

```

1.  init
2.  do {
3.      do {
4.          next2
5.      } while not last2
6.      reinit2
7.      next1
8.  } while not last1

```

(b)

**Figure 3.1** (a) The general structure of a loopless algorithm. (b) The general structure for nested loopless algorithms.

In practice, nesting loopless algorithms will require a structure similar but not necessarily identical to that of Figure 3.1. Variations may depend the algorithm’s purpose and component algorithms’ constraints. For example, two do-while loops were used in Figure 3.1, to communicate the concept of nested loops most clearly, and to place the terminating conditions at the end of the loops where it is most intuitive to explain them. One possible variation would be to use while loops instead. Another relates to the fact that, as written, Figure 3.1(b) implies that the final call to *next1* is redundant, since no cycle of the inner algorithm follows it. In cases where a redundant call is undesirable, the code on lines 3–5 would have to be repeated following line 8.

### 3.2 Reinitialising

Regardless of the exact implementation, nesting requires a new structural element: reinitialisation, on line 6. The inner algorithm must be run through many full cycles in succession, something it is unlikely to have been designed to do. Also, by definition, a loopless algorithm’s final state differs from its initial state. Thus, additional code is needed to prepare the algorithm to run another cycle, or reinitialise it.

There are two ways to reinitialise a loopless algorithm. The first is to

*refresh* it, or return it to its initial state. The second is to *reverse* it, or change it so that it will run backwards from its final state. In either method, a reinitialisation code block is only allowed  $O(1)$  time, because it occurs between objects. Some loopless algorithms are trivial to reinitialise in  $O(1)$  time, such as GRAYLPL in Section 1.2, which required just one variable to be reset. Others, however, may be problematic, requiring  $O(n)$  adjustments.

There are a couple of techniques, however, that allow  $O(n)$  reinitialisation steps to be carried out in  $O(1)$  time per object. The first is to perform one step per object generated, thus distributing  $O(n)$  steps over  $O(n)$  objects, where the complete listing has at least that many objects. This technique is nicknamed *time-stealing*, since reinitialisation steals time during object generation. An example of this might involve some array  $a[1 \dots n]$ , of which all  $n$  elements need reinitialising. Supposing that the algorithm finishes with each  $a[i]$  at different points throughout its cycle, each  $a[i]$  can be reinitialised as soon as it is finished with. This requires tests each and every object, however, and therefore is less efficient than if reinitialisation can be carried out in one step at the end.

The second trick can be used in any circumstance, although it uses space less efficiently. It involves maintaining two copies of any array requiring  $O(n)$  reinitialisation. In any cycle of the algorithm, one copy is being used, while the other is being reinitialised, one element at a time. Each cycle, the algorithm would alter some flag keeping track of which elements are active and which are being reinitialised. An example of this might involve arrays  $a[1 \dots n]$  and  $b[1 \dots n]$ . In one cycle, array  $a$  would be used while array  $b$  was being reinitialised using time-stealing; the arrays would alternate tasks in subsequent cycles.

Examples of each of these techniques can be found in subsequent algorithms in this thesis. Both new algorithms, MIXPAR in Chapter 4 and MULTPERM in Chapter 5, use the first technique to reinitialise  $O(n)$  array elements over several iterations of the inner algorithm. Korsh & LaFollette use the second technique, maintaining two versions of Eades & McKay’s combinations algorithm, one for running forwards and the other backwards. This contributes to MULTPERM’s greater space efficiency compared to that of Korsh & LaFollette’s algorithm, as discussed in Section 5.4.

```

1.  init
2.  while not last do {
3.       $e[n] = n$ 
4.      if  $d[j]$  is 1 then  $inc(obj[j])$  else  $dec(obj[j])$ 
5.      if  $extr(obj[j])$  then {
6.           $e[j] = e[j - 1]$ ,  $e[j - 1] = j - 1$ ,  $d[j] = -d[j]$ 
7.          reinit
8.      }
9.       $j = e[n]$ 
10. }

```

**Figure 3.2** A generalised version of Williamson’s algorithm.

### 3.3 Fusing with Williamson’s Algorithm

As discussed in Section 2.2, Williamson’s algorithm is a general loopless algorithm that can itself be used as a structure for fusing, a general form for which is shown in Figure 3.3. It requires that procedures *inc* and *dec* run in  $O(1)$  time, as does function *extr*, which returns whether the given element has reached an extremal value. This offers considerable flexibility to insert existing loopless algorithms into Williamson’s structure. For example, Xiang & Ushijima modified this algorithm to produce a loopless algorithm for parenthesis strings, while Section 1.3 cites several examples of combinations algorithms being fused within Williamson’s to produce algorithms for multiset permutations.

If a loopless algorithm has been inserted as the incrementing and decrementing procedure, then the extremal value occurs when the algorithm has finished a complete cycle. Reinitialisation, discussed in the previous section, must take place by the end of these cycles, and thus nominally occurs on line 7. As discussed in the previous section,  $O(n)$  reinitialisation steps may have been distributed over prior generation steps. A notable difference is that the nesting structure from Section 3.1 results in one full cycle of the inner algorithm per iteration of the outer algorithm, while fusing with Williamson’s algorithm results in one iteration of the inner algorithm per iteration of the

outer algorithm, although several full cycles of the inner algorithm still occur during the full cycle of the outer algorithm.

The framework described here, comprising the nesting structure, the generalised Williamson’s algorithm, and the reinitialisation techniques, is applied to two combinatorial generation problems in the next two chapters. A new algorithm for mixed parenthesis strings is developed in Chapter 4. It nests algorithms for parenthesis strings and Gray codes, and refreshes its inner Gray algorithm. A new algorithm for multiset permutations is developed in Chapter 5. It fuses an algorithm for combinations within Williamson’s algorithm, and reverses the inner combinations algorithm. Both new algorithms make use of the time-stealing technique.

## Chapter IV

### Generating Mixed Parenthesis Strings

This chapter applies the general framework from Chapter 3 to the area of parenthesis strings. A new set of combinatorial objects is introduced: *mixed parenthesis strings*, which comprise parentheses of different types. A loopless algorithm for generating mixed parenthesis strings is developed by fusing algorithms for parenthesis strings and binary strings. This algorithm required the development of a novel method for finding the positions of right parentheses in  $O(1)$  time, covered in Section 4.2.

#### 4.1 *Mixed Parenthesis Strings*

Mixed parenthesis strings comprising parentheses of two types are produced by grammars of the form  $S \rightarrow \epsilon \mid (S) \mid [S] \mid SS$ . This means pairs must be well nested, and both parentheses in any pair must have the same type. For example,  $( ) [ ]$  and  $( [ ] )$  are valid mixed parenthesis strings, while  $( ] [ )$  and  $( [ ) ]$  are not. We consider only two types of parenthesis in this thesis, but it would be easy to extend both the concept and the algorithm beyond binary. A listing of mixed parenthesis strings is given in Figure 4.1; the remainder of this chapter develops the new algorithm for generating them.

A mixed parenthesis string can be represented by a parenthesis string and a binary string. The parenthesis string describes the order of left and right parentheses, while the binary string describes the pairs' types. For example, the mixed parenthesis string  $( ) [ ]$  can be represented by the parenthesis string  $( ) ( )$  and the binary string 01, where 0 and 1 stand for round and square parentheses respectively. The  $i$ th bit of the binary string is the type of the  $i$ th pair, where the  $i$ th pair contains the  $i$ th left parenthesis counted



1. ( ) ( ) ( )	9. ( ) ( ( ) )	17. ( ( ) ( ) )
2. ( ) ( ) [ ]	10. ( ) ( [ ] )	18. ( ( ) [ ] )
3. ( ) [ ] [ ]	11. ( ) [ [ ] ]	19. ( [ ] [ ] )
4. ( ) [ ] ( )	12. ( ) [ ( ) ]	20. ( [ ] ( ) )
5. [ ] [ ] ( )	13. [ ] [ ( ) ]	21. [ [ ] ( ) ]
6. [ ] [ ] [ ]	14. [ ] [ [ ] ]	22. [ [ ] [ ] ]
7. [ ] ( ) [ ]	15. [ ] ( [ ] )	23. [ ( ) [ ] ]
8. [ ] ( ) ( )	16. [ ] ( ( ) )	24. [ ( ) ( ) ]
25. ( ( ( ) ) )	33. ( ( ) ) ( )	
26. ( ( [ ] ) )	34. ( ( ) ) [ ]	
27. ( [ [ ] ] )	35. ( [ ] ) [ ]	
28. ( [ ( ) ] )	36. ( [ ] ) ( )	
29. [ [ ( ) ] ]	37. [ [ ] ] ( )	
30. [ [ [ ] ] ]	38. [ [ ] ] [ ]	
31. [ ( [ ] ) ]	39. [ ( ) ] [ ]	
32. [ ( ( ) ) ]	40. [ ( ) ] ( )	

**Figure 4.1** MIXPAR algorithm output for  $n = 3$ . Each column of eight strings represents a complete Gray code cycle. Parenthesis strings change between columns.

from the beginning of the string.

An algorithm for generating mixed parenthesis strings, then, nests algorithms for generating parenthesis strings and binary strings. In this way, either all binary strings are generated for each parenthesis string, or vice versa. For example, the eight mixed parenthesis strings of 2 pairs could be generated in the following way. First, one of the two parenthesis strings of size  $2n$  is generated, say  $( ) ( )$ . For this parenthesis string, the four binary strings of size  $n = 2$  — 00, 01, 10 and 11 — are generated. This pattern is then repeated for the other parenthesis string,  $( ( ) )$ . Code could be added to translate from parenthesis strings and binary strings to mixed

par	bin	mixpar	bin	par	mixpar
...	...	...	...	...	...
( ( ) ( ) )	000	( ( ) ( ) )	011	( ) ( ) ( )	( ) [ ] [ ]
	001	( ( ) [ ] )		( ) ( ( ) )	( ) [ [ ] ]
	011	( [ ] [ ] )		( ( ) ( ) )	( [ ] [ ] )
	010	( [ ] ( ) )		( ( ( ) ) )	( [ [ ] ] )
	110	[ [ ] ( ) ]		( ( ) ) ( )	( [ ] ) [ ]
...	...	...	...	...	...
(a)			(b)		

**Figure 4.2** Excerpts from listings of mixed parenthesis strings generated with: (a) the binary strings algorithm nested inside, and (b) the parenthesis strings algorithm nested inside. The former is easier to implement.

parenthesis strings if required. As hinted earlier, extending the concept to  $k$  types of parentheses would simply be a matter of implementing a  $k$ -ary strings algorithm instead of binary.

Implementing a similar, loopless algorithm using the nesting framework from the previous chapter raises two issues. The first is that both component algorithms must operate directly on the mixed parenthesis string in  $O(1)$  time. A loopless algorithm cannot afford  $O(n)$  time to translate from parenthesis strings and binary strings to mixed parenthesis strings. The second issue is that the inner algorithm in the nesting must be reinitialised at the end of each full cycle. It is more efficient and more elegant if this can be done in  $O(1)$  time, rather than by one of the time-stealing techniques discussed in the previous chapter.

Regarding the first issue, both component algorithms can operate directly on the mixed parenthesis string in  $O(1)$  time when the algorithm for binary strings is nested inside that for parenthesis strings. In this configuration, the only required modification to the binary strings algorithm is that where it would normally invert a bit, it should invert the types of the corresponding pair of parentheses, as shown in Figure 4.2(a). Provided the positions of the parentheses in each pair are known, this operation can easily be done in  $O(1)$

par	bin	mixpar	par	bin	mixpar
...	...	...	...	...	...
( ) ( ( ) )	000	( ) ( ( ) )	( ) ( ( ) )	000	( ) ( ( ) )
...	...	...	...	...	...
	100	[ ] ( ( ) )		100	[ ] ( ( ) )
( ( ) ( ) )	000	( ( ) ( ) )	( ( ) ( ) )	100	[ ( ) ( ) ]
...	...	...	...	...	...
(a)			(b)		

**Figure 4.3** Excerpts from listings of mixed parenthesis strings generated with: (a) the inner algorithm refreshing at the end of each cycle, and (b) the inner algorithm reversing. The former is easier to implement.

time. The opposite configuration, with the algorithm for parenthesis strings nested inside that for binary strings, is more complicated. The parenthesis strings algorithm usually swaps exactly two parentheses, but it would have to be modified to sometimes swap more than two to balance the different types. For example, between the second and third output lines of Figure 4.2(b), the parenthesis strings algorithm would normally swap the second and third parentheses; for the mixed parenthesis string, however, these parentheses are of different types, so at least one extra operation is necessary. It is not obvious that this extra work can be done in  $O(1)$  time, let alone how.

Addressing the second issue, the most advantageous way to reinitialise an algorithm for binary strings is to *refresh* one that generates them in Gray code order, such as GRAYLPL (Algorithm 2) from Section 1.2. In Gray code order, the last object differs by only one bit from the first. For mixed parenthesis strings, this means the final string in a Gray cycle differs from the first by only the type of one pair. Furthermore,  $j$  is the only other variable that needs refreshing (since the algorithm refreshes  $e[n]$  at the beginning of each iteration). Thus, such an algorithm can easily be refreshed to its initial state in  $O(1)$  time. The big advantage of refreshing is that all parentheses are of the same type when it is the turn of the algorithm for parenthesis strings to operate. As can be seen from Figure 4.3(a), this means the parenthesis

strings algorithm can just operate as usual. Reversing, on the other hand, removes the need to refresh any parentheses, but it would require that the algorithm for parenthesis strings take into account a parenthesis pair of a different type, as shown in Figure 4.3(b). As mentioned in the last paragraph, it is not obvious that this sort of modification can be achieved within  $O(1)$  time.

## 4.2 Finding Right Parentheses

A final issue, then, is that of maintaining the positions of the parentheses in each pair, so that the algorithm for binary strings can alter the types of pairs in place. No loopless algorithm for parenthesis strings maintains this information completely, but that of Xiang & Ushijima does maintain the positions of the left parentheses correctly at all times. From the positions of left parentheses, the following method can find the position of each right parenthesis in  $O(1)$  time.

Let  $l_i$  be the position of the  $i$ th left parenthesis, and let  $r_i$  be the position of the *partner* of the  $i$ th left parenthesis (i.e. *not* simply the  $i$ th right parenthesis). For example, for the mixpar  $(( ( ) ( ) )$ ,  $l_2 = 2$  and  $r_2 = 3$ . By definition,  $r_n$  must be adjacent to  $l_n$ , since no left parentheses can occur after  $l_n$ . Thus, the equation for finding  $r_n$  is:

$$r_n = l_n + 1 \quad (4.1)$$

It follows that  $l_{n-1}$  and  $r_{n-1}$  must either be similarly adjacent, or else nested around the  $n$ th pair. This method for finding right parentheses is based on the idea that each  $r_i$ , starting from  $r_n$  and working to  $r_1$ , must be either adjacent to  $l_i$  or else nested around a substring that ends with some  $r_j$ , where  $j > i$  and so  $r_j$  is known. Thus, information about known substrings must be maintained.

Array  $s$  is initialised such that  $s_i = i$  for  $1 \leq i \leq 2n$ . Following this initialisation, each  $s_{l_i}$  will eventually store the position immediately after the longest well formed substring that begins at  $l_i$ . For example, for the mixpar  $(( ( ( ) ) )$ ,  $l_2 = 2$ ; initially  $s_{l_2} = s_2 = 2$ , but after processing  $s_{l_2}$

will store 6. From the base step of (4.1), each successive  $s_{l_i}$  and  $r_{i-1}$  can be found in constant time by working backwards  $n$  to 1 as follows.

If there is no  $l_j$  immediately after  $r_i$ , then the substring beginning at  $l_i$  ends at  $r_i$ , and  $s_{r_i+1}$  will not have changed since initialisation (described in the preceding paragraph). On the other hand, if there is some  $l_j$  immediately after  $r_i$ , then the substrings beginning at  $l_i$  and  $l_j$  end in the same position; because the method works backwards from the  $n$ th pair,  $s_{l_j}$  will already have been set correctly. Thus, we derive an unconditional equation for  $s_{l_i}$  independent of  $j$ :

$$\begin{aligned} s_{l_i} &= \begin{cases} r_i + 1 & = s_{r_i+1} \text{ iff } r_i + 1 \neq l_j \\ s_{l_j} & = s_{r_i+1} \text{ iff } r_i + 1 = l_j \end{cases} \\ &= s_{r_i+1} \end{aligned} \quad (4.2)$$

Similarly for  $r_i$ , if the  $(i+1)$ th left parenthesis is not immediately after  $l_i$ , then  $r_i$  must be, and  $s_{l_{i+1}}$  will not have changed since initialisation. Conversely, if the  $i$ th and  $(i+1)$ th left parentheses are adjacent, then  $r_i$  must be immediately after the substring starting at  $l_{i+1}$ . Because we are working backwards from the  $n$ th pair,  $s_{l_{i+1}}$  will already have been set correctly. Thus, we derive an unconditional equation for  $r_i$ :

$$\begin{aligned} r_i &= \begin{cases} l_i + 1 & = s_{l_{i+1}} \text{ iff } l_i + 1 \neq l_{i+1} \\ s_{l_{i+1}} & = s_{l_{i+1}} \text{ iff } l_i + 1 = l_{i+1} \end{cases} \\ &= s_{l_{i+1}} \end{aligned} \quad (4.3)$$

Thus, using (4.1), (4.2) and (4.3), each right parenthesis can be found in  $O(1)$  time, just as each is needed during the first half of the Gray cycle. Each  $s_{l_i}$  can be reset during the second half of the Gray cycle, using the time-stealing technique discussed in the previous chapter. Combined with Xiang & Ushijima's algorithm, which is covered in the next section, this mathematics provides the positional information needed to generate mixed parenthesis strings.

	<i>par</i>	<i>l</i>	<i>j</i>
1.	( ) ( ) ( ) ( )	1 3 5 7	4
2.	( ) ( ) ( ( ) )	1 3 5 6	3
3.	( ) ( ( ) ) ( )	1 3 4 6	4
4.	( ) ( ( ( ) ) )	1 3 4 5	4
5.	( ) ( ( ) ) ( )	1 3 4 7	2
6.	( ( ) ) ( ) ( )	1 2 4 7	4
7.	( ( ) ) ( ( ) )	1 2 4 5	4
8.	( ( ) ) ( ) ( )	1 2 4 6	3
9.	( ( ( ) ) ) ( )	1 2 3 6	4
10.	( ( ( ) ) ) ( )	1 2 3 5	4
11.	( ( ( ( ) ) ) )	1 2 3 4	4
12.	( ( ( ) ) ) ( )	1 2 3 7	3
13.	( ( ) ) ( ) ( )	1 2 5 7	4
14.	( ( ) ) ( ( ) )	1 2 5 6	1

**Figure 4.4** Parenthesis strings of  $n = 4$  pairs generated by Xiang & Ushijima’s algorithm

### 4.3 Xiang & Ushijima’s Algorithm

Xiang & Ushijima’s loopless algorithm generates well-formed parenthesis strings, which can be derived from the grammar  $S \rightarrow \epsilon \mid (S) \mid SS$ . The algorithm is given in Algorithm 4, and a complete C program in Appendix D. A sample output for  $n = 4$  pairs of parentheses is shown in Figure 4.4. The algorithm is based on Williamson’s algorithm for variations in Gray code order, covered in Sections 2.2 and 3.3, with modifications to satisfy the constraints of parenthesis strings. Where Williamson’s algorithm operates on elements in array  $v$ , allowing each to vary between 0 and some  $r[i] - 1$  in increments of 1, Xiang & Ushijima’s algorithm operates on the positions of left parentheses, maintained in array  $l$ , and whose travel differs in two important ways.

The first such difference is that in Xiang & Ushijima’s algorithm the extremal values for each element dynamically change during execution. The

	<i>par</i>	<i>l</i>
	...	...
96.	( ( ( ( <u> </u> ) ) ) ) <u> </u> ( ) )	1 2 3 4 <u>5</u> 10
97.	( ( ( ( ( ) ) ) ) <u> </u> <u> </u> ) )	1 2 3 4 5 <u>9</u>
98.	( ( ( ( ( ) ) ) <u> </u> <u> </u> ) ) )	1 2 3 4 5 <u>8</u>
99.	( ( ( ( ( ) ) <u> </u> <u> </u> ) ) ) )	1 2 3 4 5 <u>7</u>
100.	( ( ( ( ( <u> </u> <u> </u> ) ) ) ) )	1 2 3 4 5 <u>6</u>
101.	( ( ( ( ( <u> </u> ) ) ) ) ) <u> </u> )	1 2 3 4 5 <u>11</u>
102.	( ( ( ( <u> </u> <u> </u> ) ) ) ) ( )	1 2 3 4 <u>6</u> 11
103.	( ( ( ( ) ( <u> </u> ) ) ) ) <u> </u> )	1 2 3 4 6 <u>7</u>
104.	( ( ( ( ) ( <u> </u> <u> </u> ) ) ) )	1 2 3 4 6 <u>8</u>
105.	( ( ( ( ) ( ) ) <u> </u> <u> </u> ) ) )	1 2 3 4 6 <u>9</u>
106.	( ( ( ( ) ( ) ) ) <u> </u> <u> </u> ) )	1 2 3 4 6 <u>10</u>
107.	( ( ( ( ) ) <u> </u> <u> </u> ) ) ( ) )	1 2 3 4 <u>7</u> 10
	...	...

**Figure 4.5** Excerpt from Xiang & Ushijima’s algorithm output for  $n = 6$ .

minimal position for any left parenthesis except the first adjacent to its leftward neighbour (the first left parenthesis must occupy position 1). Thus, the minimal position of the  $j$ th left parenthesis is  $l[j - 1] + 1$ . For example, in the string ( ) ( ( ) ) or  $l = \{1, 3, 4\}$ , element  $l[3] = 4$  is minimal, while element  $l[2] = 3$  is not. In the opposite direction, the maximal position for some  $j$ th left parenthesis is immediately after a substring of  $j - 1$  pairs. Thus, the maximal position of the  $j$ th left parenthesis is  $l[j] = 2j - 1$ . For example, all elements in the string ( ) ( ) ( ) or  $l = \{1, 3, 5\}$  are maximal, since no left parenthesis can be moved rightwards without producing an invalid string. In theory, the maximal position of a left parenthesis would also be bounded by its rightward neighbour, but Xiang & Ushijima’s algorithm avoids such interference by use of a movement mechanism covered in the next paragraph.

The second important difference from Williamson’s algorithm is that elements generally do not vary from their minimum to their maximum in increments of 1, or vice versa. Instead, left parentheses begin and end all travels

**Algorithm 4** Xiang & Ushijima's loopless algorithm for parenthesis strings.

```

/* Main. */
1.  init
2.  read  $n$ 
3.  for  $i = 1$  to  $2n$  by  $2$  do  $par[i] = '(', par[i + 1] = ')'$ 
4.  for  $i = 1$  to  $n$  do  $l[i] = 2i - 1$ 
5.  for  $i = 1$  to  $n$  do  $d[i] = 1$ 
6.  for  $i = 1$  to  $n$  do  $e[i] = i$ 
7.   $j = n$ 
8.  print  $par$ 
9.  while  $j$  is not 1 do {
10.     next
11.     print  $par$ 
12. }

```

(continued)

in either their maximal or *submaximal* (one less than maximal) positions, that is  $l[j] = 2j - 1$  or  $2j - 2$ . A consequence of this is that the  $j$ th left parenthesis finishes moving above the maximal position of the  $(j - 1)$ th left parenthesis, which is  $l[j - 1] = 2(j - 1) - 1 = 2j - 3$ . This is why, as mentioned in the preceding paragraph, rightward neighbours never interfere with the maximal position of a left parenthesis. Besides moving by increments of 1, the algorithm also allows a left parenthesis to jump from its maximal to its minimal position and vice versa. This is how all positions from minimum to maximum can be travelled through while always starting and finishing at the maximal and submaximal positions (or vice versa). If the  $j$ th left parenthesis begins in its maximal position, its first move is to jump down to its minimal position, before climbing back up to its submaximal position by steps of 1. In the opposite case, when it begins in its submaximal position, it climbs down to its minimal position by steps of 1, before jumping up to its maximal position. The former is nominally moving in the positive direction,



**Algorithm 4** (continued)

```

/* Looplessly generates the next parenthesis string. */
13. procedure next {
14.      $e[n] = n$ 
15.      $temp = l[j]$ 
16.     if  $d[j]$  is 1 then {
17.         if  $l[j]$  is  $2j - 1$  then  $l[j] = l[j - 1] + 1$ 
18.         else add 1 to  $l[j]$ 
19.     } else {
20.         if  $l[j]$  is  $l[j - 1] + 1$  then  $l[j] = 2j - 1$ 
21.         else subtract 1 from  $l[j]$ 
22.     }
23.     swap  $par[temp]$  and  $par[l[j]]$ 
24.     if  $l[j] > 2j - 3$  then {
25.          $d[j] = -d[j]$ 
26.          $e[j] = e[j - 1]$ 
27.          $e[j - 1] = j - 1$ 
28.     }
29.      $j = e[n]$ 
30. }
```

while the latter is in the negative direction.

For example, Figure 4.5 shows an excerpt from Xiang & Ushijima’s listing for  $n = 6$ , in which the two parentheses that swapped positions from the previous line are underlined. On line 96, the fifth parenthesis has just jumped from its maximal position of 9 down to its minimal position of 5; therefore it is moving in the positive direction. On lines 97–101, the sixth parenthesis, moving in the negative direction, steps down from its submaximal (one below maximal) position of 10 to its minimal position of 6, before jumping up to its maximal position of 11. On line 102, the fifth parenthesis, moving in the positive direction, begins stepping up towards its submaximal position. On lines 103–106, the sixth parenthesis, now moving in the positive

direction, jumps from its maximal position of 11 to its minimal position of 7, before stepping up to its submaximal position of 10. Finally, on line 107, the fifth parenthesis takes another step up in the positive direction. Note that regardless of whether the sixth parenthesis was moving in the negative (lines 97–101) or positive direction (103–106), it finishes above the maximal position of the fifth parenthesis. Thus, even though the sixth parenthesis moves through all positions between movement steps of the fifth parenthesis, its high finishing position means it never interferes. The process continues with elements being selected in the same order as Williamson’s algorithm until the first element is selected and the algorithm terminates.

Xiang & Ushijima’s algorithm is an ideal component algorithm for fusing. Its few variables and concise code add little clutter to the fused algorithm. Furthermore, it is an easily understood module, since it is based on the familiar model of Williamson’s algorithm. The features familiar from Williamson’s algorithm include: variable  $j$  and array  $e$ , for loopless element selection as if by recursive tree traversal; array  $d$ , which maintains the direction of each element. The next section presents the fused algorithm, of which Xiang & Ushijima’s algorithm is a component.

#### **4.4 Algorithm MIXPAR**

MIXPAR, shown in Algorithm 5, is a loopless algorithm for generating mixed parenthesis strings. It builds on many of the ideas presented thus far in this thesis. First, it represents a successful application of the general framework developed in Chapter 3, its main code chunk (lines 1–10) strongly resembling the original general structure for nesting from Figure 3.1(b). Second, it implements the fused algorithm devised in Section 4.1, generating mixed parenthesis strings by fusing algorithms for parenthesis strings and binary strings. Those component algorithms are GRAYLPL from Section 1.2 and XUPAR, which was covered in Section 4.3. Third, it includes the mechanism for finding right parentheses in  $O(1)$  time, using arrays  $r$  and  $s$ , presented in Section 4.2. This method itself borrows the time-stealing technique from Section 3.2 to reinitialise array  $s$ . Thus, MIXPAR is a culmination of the ideas presented so far.

**Algorithm 5** MIXPAR, a loopless algorithm for mixed parenthesis strings

```

    /* Main. */
1.  init
2.  do {
3.      do {
4.          print par
5.          nextGray
6.      } while jj is not 0
7.      print par
8.      reinit
9.      nextPar
10. } while e[1] is not 0

    /* Initialisation. */
11. procedure init {
12.     read n
13.     for i = 1 to 2n by 2 do par[i] = '(', par[i + 1] = ')'
14.     for i = 1 to n do l[i] = 2i - 1
15.     for i = 0 to n do e[i] = i, ee[i] = i
16.     for i = 1 to n do d[i] = 1
17.     for i = 1 to 2n + 1 do s[i] = i
18.     j = n, jj = n
19.     t = n
20. }
```

(continued)

A sample output for MIXPAR, generating all mixed parenthesis strings with  $n = 4$  pairs, is given in Figure 4.1 on page 23. The output has been broken into columns of eight strings. Within each column, pairs change types according the Gray cycle. Between columns, the parenthesis strings change by Xiang & Ushijima's algorithm. A complete C program for MIXPAR is given in Appendix E.

**Algorithm 5** (continued)

```
/* Looplessly generates the next string by Gray code. */
21. procedure nextGray {
22.      $ee[n] = n$ 
23.     if  $jj$  is  $t$  then find
24.     change type of  $par[l[jj]]$  and  $par[r[jj]]$  (round  $\leftrightarrow$  square)
25.      $ee[jj] = ee[jj - 1]$ 
26.      $ee[jj - 1] = jj - 1$ 
27.      $jj = ee[n]$ 
28. }

/* Finds the right parenthesis in the  $jj$ th pair. */
29. procedure find {
30.     if  $par[l[1]]$  is '(' then {
31.          $r[jj] = s[l[jj] + 1]$ 
32.         if  $jj$  is not 1 then {
33.              $s[l[jj]] = s[r[jj] + 1]$ 
34.             subtract 1 from  $t$ 
35.         } else  $t = 2$ 
36.     } else {
37.          $s[l[jj]] = l[jj]$ 
38.         add 1 to  $t$ 
39.     }
40. }
```

(continued)

Most of the variables in MIXPAR are inherited from its component algorithms, with the exception of arrays  $r$  and  $s$  and variable  $t$ . The use of  $r$  and  $s$  to find right parentheses was covered in Section 4.2. Variable  $t$  is used to keep track of which elements in  $r$  and  $s$  to update next. It counts down from  $n$  to 1 during the first half of the Gray cycle, then back up to  $n$  during the second half. Thus, in the first half,  $jj = t$  signifies that  $r[jj]$  is needed

**Algorithm 5** (continued)

```

/* Reinitialises the Gray code algorithm. */
41. procedure reinit {
42.      $par[l[1]] = '(', par[r[1]] = ')'$ 
43.      $jj = n$ 
44.      $t = n$ 
45. }

/* Looplessly generates the next string by Xiang & */
/* Ushijima's algorithm. */
46. procedure nextPar {
47.      $e[n] = n$ 
48.      $temp = l[j]$ 
49.     if  $d[j]$  is 1 then {
50.         if  $l[j]$  is  $2j - 1$  then  $l[j] = l[j - 1] + 1$ 
51.         else add 1 to  $l[j]$ 
52.     } else {
53.         if  $l[j]$  is  $l[j - 1] + 1$  then  $l[j] = 2j - 1$ 
54.         else subtract 1 from  $l[j]$ 
55.     }
56.     swap  $par[temp]$  and  $par[l[j]]$ 
57.     if  $l[j] > 2j - 3$  then {
58.          $d[j] = -d[j]$ 
59.          $e[j] = e[j - 1]$ 
60.          $e[j - 1] = j - 1$ 
61.     }
62.      $j = e[n]$ 
63. }
```

for the first time and therefore must be found. Except when  $jj = 1$ , element  $s[l[jj]]$  must also be found, to assist the subsequent finding of  $r[jj - 1]$ . In the second half,  $jj = t$  signifies that  $s[l[jj]]$  should be reinitialised.

Procedure *find* performs the finding and reinitialising for arrays  $r$  and  $s$

(lines 29–40). It determines which half of the Gray cycle it is in by the type of the first parenthesis: round means first half (line 30). Right parenthesis  $r[jj]$  is found using formula (2) from Section 4.2 (line 31). If  $jj \neq 1$ , then  $r[jj - 1]$  will need to be found next, so  $s[l[jj]]$  is found using formula (3) from Section H, and  $t$  is decremented by 1 (lines 32–34). On the other hand, if  $jj = 1$ ,  $s[l[jj]]$  is not changed, and  $t$  is set to 2 in preparation for the second half of the Gray cycle (line 35). During the second half, each  $s[l[jj]]$  is reinitialised to  $l[jj]$ , and  $t$  is incremented by 1 each time (lines 37–38).

Procedure *reinit* reinitialises the Gray code algorithm (lines 41–45). As suggested in Section 4.1, this is a very simple operation. The first pair of parentheses must be changed from square to round. The Gray cycle dictates that the first pair will always finish square, while all others will be round. Second,  $jj$  and  $t$  are reset from their final values of 0 and  $n + 1$  to their initial values of  $n$  and  $n$  respectively. Note that *reinit* must be called before *nextPar* (lines 8–9), because the parenthesis strings algorithm relies on all pairs being the same type.

The exact implementation of algorithm MIXPAR is, as with most loopless algorithms, a compromise. It represents a trade-off between minimising running time, minimising space used, and maximising code readability. For example, one dilemma is that there must be one more cycle of binary strings than there are iterations of the parenthesis strings algorithm. This is because the first object is initialised, rather than generated by the algorithm. There are several ways in which this could be coded. The method chosen here wastes a little time and space, but makes for the most readable code. The algorithm for parenthesis strings is forced to generate one extra string, although it is not output. To enable this, array  $e$  has been extended to include index 0, which is used in the new termination condition (line 10). The benefit of these costs is that the code is kept simple, the structure of the loops reflects the design of the algorithm, and each loop has a simple condition. Time and space could be saved by repeating the block of code that generates binary strings, but that would make for convoluted code. It is worth remembering that for all loopless algorithms in this thesis, some further tweaking for time, space, and/or readability may be possible; these changes are cosmetic in that they do not affect the algorithms'  $O(1)$  time

complexity.

This chapter applied the general framework for fusing loopless algorithms to produce a new loopless algorithm for generating mixed parenthesis strings. The next chapter applies the same framework to a different problem, that of generating multiset permutations.

## Chapter V

### Generating Multiset Permutations

This chapter presents a second application of the general framework from Chapter 3, this time to the area of multiset permutations. A loopless algorithm for generating multiset permutations is developed by fusing algorithms for combinations and element selection. This approach is similar to Korsh & LaFollette's, but the algorithm presented here is simpler, smaller and faster.

#### 5.1 *Multiset Permutations*

A multiset is a set that allows repetitions. It has  $k$  distinct elements, which we assume without loss of generality to be the integers  $1, 2, \dots, k$ . Each distinct element  $i$  has a multiplicity  $m_i$ , which is the number of times it appears in the multiset. The size  $n$  of the multiset, then, is the sum of all multiplicities. For example, the multiset  $\{1, 1, 1, 2, 2, 3\}$  has  $k = 3$ ,  $m = \{3, 2, 1\}$  and  $n = 6$ .

Multiset permutations are distinct arrangements of the elements in a multiset. For example,  $\{1, 1, 2, 1, 2, 3\}$  and  $\{3, 1, 1, 2, 2, 1\}$  are two permutations of the same multiset. Note that swapping similar elements does not generate a new permutation. Thus, an algorithm that takes the permutation  $\{1, 1, 2, 1, 2, 3\}$  and swaps the first two elements to produce  $\{1, 1, 2, 1, 2, 3\}$  has generated the same permutation again. A listing of multiset permutations is given in Figure 5.1; the remainder of this chapter describes the development of the new algorithm for generating them.

An algorithm for generating multiset permutations can be based on the well known Johnson and Trotter algorithm for set permutations, as mentioned in Section 1.3. Where the Johnson-Trotter algorithm moves single selected elements through subpermutations, an algorithm for multiset permutations moves groups of selected elements through subpermutations. Thus,



1. 1 1 2 2 3	11. 2 3 2 1 1	21. 1 1 3 2 2
2. 1 2 1 2 3	12. 2 3 1 2 1	22. 1 3 1 2 2
3. 2 1 1 2 3	13. 2 1 3 2 1	23. 3 1 1 2 2
4. 2 2 1 1 3	14. 1 2 3 2 1	24. 3 2 1 1 2
5. 2 1 2 1 3	15. 1 2 3 1 2	25. 3 1 2 1 2
6. 1 2 2 1 3	16. 2 1 3 1 2	26. 1 3 2 1 2
7. 1 2 2 3 1	17. 2 3 1 1 2	27. 1 3 2 2 1
8. 2 1 2 3 1	18. 2 1 1 3 2	28. 3 1 2 2 1
9. 2 2 1 3 1	19. 1 2 1 3 2	29. 3 2 1 2 1
10. 2 2 3 1 1	20. 1 1 2 3 2	30. 3 2 2 1 1

**Figure 5.1** Multiset permutations generated by MULTPERM for inputs  $k = 3$ ,  $m = \{2, 2, 1\}$ .

the 4s can be moved as a group, then the 3s, and so on. A recursive algorithm for multiset permutations is as follows.

Let  $perm$  be a multiset permutation of  $n$  integers. Let  $subp_i$  be a sub-permutation of  $perm$  comprising all elements *greater than*  $i$ . Initially  $perm$  is the lexicographically least permutation. If  $k = 1$  then  $perm$  is the only permutation. Otherwise, the 1s are placed among  $subp_1$  in all remaining distinct ways such that the relative order of elements of  $subp_1$  is maintained, and  $subp_1$  is contiguous in the final permutation. This generates all permutations containing  $subp_1$ . If there is another  $subp_1$  of  $perm$ , it is generated recursively, and the next  $perm$  becomes this next  $subp_1$  bounded by the 1s. The 1s are now placed among this next  $subp_1$  in all remaining distinct ways, subject to the same conditions as before. This generates all permutations containing this next  $subp_1$ . This process of moving 1s through  $subp_1$ s continues until they have appeared in all distinct ways in the last  $subp_1$ . When the  $k$  integers are distinct this algorithm mimics the Johnson-Trotter algorithm.

This algorithm can be implemented looplessly, using the mechanism from Williamson's algorithm (involving array  $e$  and variable  $j$ ) to select elements in the same order as an inorder tree traversal. Grouped element movement can be achieved using a combinations algorithm, which are often visualised

as moving 0s or 1s through a binary string, for which there are many loopless candidates. A similar approach was the basis for loopless algorithms by Korsh & LaFollette, among others, as discussed in Section 2.3.

The algorithm described above has a small but significant difference from that described by Korsh & LaFollette, which ultimately led to a simpler and more efficient algorithm. They required that similar elements finish moving through a subpermutation at one end (left or right) of the subpermutation. For example, Korsh & LaFollette require the 3s would finish  $\{3, 3, 1, 2, 2\}$  or  $\{1, 2, 2, 3, 3\}$ . In the new algorithm being described here, it is only required that the subpermutation be contiguous, meaning similar elements may finish distributed over both ends. Thus, this approach also allows the 3s to finish  $\{3, 1, 2, 2, 3\}$ . Because this requirement is less strict than that used by Korsh & LaFollette, it allows a wider range of combinations algorithms to be considered.

Both algorithms share two further requirements: that their combinations algorithm maintain the relative order of 0s, and that it can be reversed in  $O(1)$  time. The first implies that, when applied to an algorithm for multiset permutations, moving the 3s through the 1s and 2s does not affect their order. For example, if the 3s moved through all possible positions beginning with  $\{1, 2, 2, 3, 3\}$  and ending with  $\{3, 3, 1, 2, 2\}$ , the relative order of the subpermutation  $\{1, 2, 2\}$  remains unaffected. Without this property it would not be obvious that the algorithm had generated all distinct permutations. The second requirement is due to the fact that the combinations algorithm must be run many times in succession, and that refreshing such an algorithm used for multiset permutations would require  $O(k)$  time. For example, refreshing the 5 from  $\{5, 1, 2, 3, 4\}$  back to  $\{1, 2, 3, 4, 5\}$  requires each of the  $k$  elements to be shuffled along one position, which cannot be done in  $O(1)$  time in a simple representation. (Note that Korsh & Lipschutz use a linked list representation to perform such operations in constant time, but this violates Ehrlich's criteria for loopless algorithms.)

Finally, the combination algorithm's transpositions must be able to be translated to the multiset in  $O(1)$  time. Korsh & LaFollette showed that  $O(n)$ -distance transpositions could be managed with little extra bookkeeping if all elements being jumped were similar. For example, the transition

from  $\{1, 1, 1, 1, 2\}$  to  $\{2, 1, 1, 1, 1\}$  can be performed in  $O(1)$  time since the intermediate 1s are unaffected. As mentioned in the preceding paragraph, if the elements differ it would require  $O(k)$  time. The approach described in this thesis is even simpler, because it uses a combinations algorithm whose transpositions are limited to  $O(1)$  distance.

## 5.2 Chase's Algorithm

In this thesis, Chase's (1989) algorithm for combinations can be thought of as an interface providing several functionalities, the most important of which is that it offers a means to looplessly generate combinations by 1- or 2-apart transpositions. This section covers *how* the algorithm works, such that it can later be used as a component with its properties. *Why* the algorithm works, on the other hand, is very difficult and outside the scope of this thesis. Readers seeking such a proof should refer to Chase's original journal article.

Chase's loopless algorithm for generating combinations by 1- or 2-apart transpositions is shown in Algorithm 6. Notably, it is the one loopless algorithm given in this thesis that does not follow the structure of Williamson's algorithm. It operates on an in-place combination, maintained in array  $c$ . For example, a combination containing the first, third and fourth elements of some set would be represented as  $c = \{1, 3, 4\}$ , where the elements are arranged conventionally in increasing order. A listing of combinations generated by Chase's algorithm is shown in Figure 5.2.

The presentation of Chase's algorithm shown here differs markedly from

	<i>comb</i>	<i>z</i>		<i>comb</i>	<i>z</i>		<i>comb</i>	<i>z</i>
1.	1 2 3 4	5	6.	1 2 5 6	3	11.	1 4 5 6	2
2.	1 2 3 5	4	7.	1 3 5 6	2	12.	1 3 4 6	2
3.	1 3 4 5	2	8.	2 3 5 6	1	13.	2 3 4 6	1
4.	2 3 4 5	1	9.	3 4 5 6	1	14.	1 2 4 6	3
5.	1 2 4 5	3	10.	2 4 5 6	1	15.	1 2 3 6	4

**Figure 5.2** Combinations generated by Chase's algorithm for  $n = 4$  out of  $r = 6$ .

the original publication, which is included as Appendix F. The original version is highly optimised for speed, resulting in code that is less readable. For example, the modulo operator is used in place of several conditional statements, obscuring the algorithm’s decision-making process. In the version presented here, with those conditional statements restored, it is now clear that the many nested if-then-else statements are determining which of ten movements to apply in a given iteration. Further benefits of the presentation in Algorithm 6, as will be discussed in this section, is that the algorithm stops after the final combination, rather than generating one too many, and that the algorithm can be easily reversed. Also, variable  $z$  is now maintained perfectly throughout the algorithm, a property that was sacrificed for speed in the original format.

All changes in Chase’s algorithm are based around variable  $z$ , which maintains the position of the first nonminimal element in  $c$ . For the example combination just given, the position of the first nonminimal element  $z = 2$ . This is because  $c[1] = 1$  is minimal, but  $c[2] = 3$  is not, since  $c[2]$  could instead hold 2. In other words,  $z = i$  where  $i$  is the smallest value for which  $c[i] \neq i$ . Thus, in bit-vector representation, position  $z$  holds the first 0 with which a nearby 1 can be transposed. Algorithm 6 shows that all changes occur within a distance of two positions from  $z$ .

Chase’s algorithm has two output variables,  $x$  and  $y$ , which at the end of each iteration hold the outgoing and incoming elements respectively. Thus, if the combination  $c = 1, 3, 4$  changes to  $c = 1, 2, 4$ , then  $x = 3$  and  $y = 2$ . These are useful for the multiset permutations algorithm, since they correspond to the positions in the permutation that must be transposed.

As originally published, Chase’s algorithm tested the condition  $y > n$  to determine when it had generated beyond the last object. Here, the test looks at the elements of the combination for a final configuration, meaning no invalid combination must be generated. If  $n = 1$ , or if  $r$  is even, then the final configuration will consist of the  $n$ th element being maximal and the  $(n - 1)$ th (and therefore all below it) minimal; otherwise, it consists of the  $(n - 1)$ th element (and therefore also the  $n$ th) being maximal and the  $(n - 2)$ th (and below) minimal. This version uses variable  $b$  to keep track of whether one or two elements are expected to finish maximally. The first part of reversing

**Algorithm 6** Chase’s loopless algorithm for combinations by 1- or 2-apart transpositions. Presentation has been changed significantly for readability.

```

/* Main. */
1.  init
2.  print comb
3.  while comb[n − b] is not n − b or comb[n − b + 1] is not r − b + 1 do {
4.      next
5.      print comb
6.  }

/* Initialisation. */
7.  procedure init {
8.      read n and r
9.      for i = 1 to n do comb[i] = i
10.     comb[n + 1] = 2r + 1
11.     z = n + 1
12.     Set b to 1 if r is even or n is 1, else 2
13. }

```

(continued)

Chase’s algorithm is to test instead for the starting configuration, in which the *n*th element is minimal (and therefore all elements are).

Chase’s algorithm uses functions *inc* and *adj* to determine which of ten movements to make. Function *inc*(*i*) tests whether element *comb*[*i*] is currently increasing, which is true if and only if element *comb*[*i* + 1] is odd. For example, *inc*(1) would return 1 if *comb*[2] = 5, meaning *comb*[1] is currently increasing. This property means the algorithm needs no extra array to store directions. Function *adj*(*i*) tests simply whether elements *comb*[*i*] and *comb*[*i* + 1] hold adjacent positions. For example, *adj*(*i*) would return true if, say, *comb*[1] = 4 and *comb*[2] = 5. This function is used to determine an element’s freedom to move. The second part of reversing Chase’s algorithm is to invert function *inc*.

**Algorithm 6** (continued)

```
/* Looplessly generates the next combination by selecting */
/* one of ten movements. */
14. procedure next {
15.     if z is 1 then {
16.         if inc(1) then {
17.             if adj(1) then {
18.                 if inc(2) then move(1, 1, 2)
19.                 else move(2, -1, 2)
20.             } else move(1, 1, 1)
21.         } else move(1, -1, 1)
22.     } else {
23.         if inc(z - 1) then {
24.             if z > 2 and inc(z - 2) then move(z - 2, 1, 2)
25.             else move(z - 1, 1, 1)
26.         } else {
27.             if not adj(z) then {
28.                 if inc(z) then move(z, 1, 1)
29.                 else move(z, -1, 1)
30.             } else {
31.                 if inc(z + 1) then move(z, 1, 2)
32.                 else move(z + 1, -1, 2)
33.             }
34.         }
35.     }
36. }
```

(continued)

As mentioned earlier, Chase's algorithm as presented here is simple to reverse, comprising three parts. First, the termination condition must be modified to test for what is normally the initial configuration. Second, function *inc* must be inverted, that is so that *inc*(*i*) returns true iff *comb*[*i*] is

**Algorithm 6** (continued)

```

    /* Returns whether comb[i] is increasing. */
37.  function inc(i) {
38.      return comb[i + 1] is odd
39.  }

    /* Returns whether comb[i] and [i+1] are adjacent. */
40.  function adj(i) {
41.      return comb[i] + 1 is comb[i + 1]
42.  }

    /* Moves an element for a given position, direction and span. */
43.  procedure move(p, d, s) {
44.      x = comb[p]
45.      y = x + s × d
46.      comb[p] = x + d
47.      comb[p + d(s − 1)] = y
48.      if comb[z] is z then add s to z
49.      else if comb[z − 1] is not z − 1 then subtract s from z
50.  }
```

even. Finally, the algorithm must be initialised to what is normally the final configuration for  $n$  and  $r$ , which was described earlier. The first two modifications can easily be made in  $O(1)$  time, making Chase's algorithm ideal for running continuously forwards and backwards inside a Williamson's algorithm structure.

Figure 5.2 (page 41) shows output for Chase's algorithm for inputs  $n = 4$  out of  $r = 6$ . Variable  $z$  has also been shown for clarity. Note that all changes are by one- or two-apart transpositions, of which examples occur on lines 1–2 and 2–3 respectively. The algorithm is used as a component within the a new algorithm for multiset permutations in the following section.

**Algorithm 7** MULTPERM, a new loopless algorithm for generating multiset permutations.

```

    /* Main. */
1.  init
2.  print
3.  while  $j$  is not  $k$  do {
4.      nextWill
5.      print
6.  }

    /* Looplessly selects the next  $j$  by Williamson's algorithm. */
7.  procedure nextWill {
8.       $e[1] = 1$ 
9.      nextChase
10.     if  $\text{comb}[j][m[j] - b[j] + 1]$  is  $r[j] - b[j] + 1$ 
11.     and  $\text{comb}[j][m[j] - b[j]]$  is  $m[j] - b[j]$ 
12.     or  $\text{comb}[j][m[j]]$  is  $m[j]$  then {
13.          $e[j] = e[j + 1]$ 
14.          $e[j + 1] = j + 1$ 
15.          $d[j] = -d[j]$ 
16.          $a[j] = j + 1$ 
17.     }
18.      $j = e[1]$ 
19. }

```

(continued)

### 5.3 Algorithm MULTPERM

MULTPERM, shown in Algorithm 7, is a loopless algorithm for generating multiset permutations in linear space using only arrays. Like MIXPAR, it relies on much of the earlier material in this thesis. Again, it is an application of the fusion framework developed in Chapter 3. This time, it is a modification of Williamson's algorithm, as discussed in Section 3.3, which can be



**Algorithm 7** (continued)

```

/* Initialisation. */
20. procedure init {
21.   read  $k$ 
22.   for  $i = 1$  to  $k$  do read  $m[i]$ 
23.   set  $n$  to the sum of  $m[1 \dots k]$ 
24.   for  $i = 1$  to  $k$  do set  $o[i]$  to the sum of all  $m[< i]$ 
25.   for  $i = 1$  to  $k$  do set  $r[i]$  to the sum of all  $m[\geq i]$ 
26.   for  $i = 1$  to  $k$  do {
27.     for  $j = 1$  to  $m[i]$  do  $perm[j + o[i]] = i$ 
28.   }
29.   for  $i = 0$  to  $k + 1$  do  $e[i] = i$ 
30.   for  $i = 1$  to  $k$  do  $d[i] = 1$ 
31.   for  $i = 1$  to  $k$  do {
32.     for  $j = 1$  to  $m[i]$  do  $comb[i][j] = j$ 
33.      $comb[i][m[i] + 1] = 2r[i] + 1$ 
34.      $z[i] = m[i] + 1$ 
35.   }
36.   for  $i = 1$  to  $k - 1$  do {
37.      $a[i] = i + 1$ 
38.     if  $m[i] > 1$  and  $r[i] \bmod 2$  is 1 then  $b[i] = 2$ 
39.     else  $b[i] = 1$ 
40.   }
41.    $j = 1$ 
42. }

```

(continued)

seen from its procedure *nextWill* (lines 7–19). It implements the fused algorithm designed in Section 5.1 by fusing Chase’s algorithm for combinations, covered in Section 5.2, with Williamson’s algorithm for variations in Gray code order, from Section 2.2. It also makes use of the time-stealing technique discussed in Section 3.2.

**Algorithm 7** (continued)

```

/* Looplessly generates the next permutation by Chase's algorithm. */
43. procedure nextChase {
44.   if  $z[j]$  is 1 then {
45.     if  $inc(1)$  then {
46.       if  $adj(1)$  then {
47.         if  $inc(2)$  then  $move(1, 1, 2)$ 
48.         else  $move(2, -1, 2)$ 
49.       } else  $move(1, 1, 1)$ 
50.     } else  $move(1, -1, 1)$ 
51.   } else {
52.     if  $inc(z[j] - 1)$  then {
53.       if  $z[j] > 2$  and  $inc(z[j] - 2)$  then  $move(z[j] - 2, 1, 2)$ 
54.       else  $move(z[j] - 1, 1, 1)$ 
55.     } else {
56.       if not  $adj(z[j])$  then {
57.         if  $inc(z[j])$  then  $move(z[j], 1, 1)$ 
58.         else  $move(z[j], -1, 1)$ 
59.       } else {
60.         if  $inc(z[j] + 1)$  then  $move(z[j], 1, 2)$ 
61.         else  $move(z[j] + 1, -1, 2)$ 
62.       }
63.     }
64.   } /* Values for  $x$  and  $y$  were produced by  $move$ . */
65.    $perm[x + o[j]] = perm[y + o[j]], perm[y + o[j]] = j$ 
66.   if  $a[j] < k$  then {
67.      $o[a[j]] = o[a[j]] - b[j] \times d[j]$ 
68.      $a[j] = a[j] + 1$ 
69.   }
70. }
```

(continued)

**Algorithm 7** (continued)

```

/* Moves an element for a given position, direction and span. */
/* Produces new values for global variables x and y. */
71. procedure move(p, d, s) {
72.     x = comb[j][p]
73.     y = x + s × d
74.     comb[j][p] = x + d
75.     comb[j][p + d(s − 1)] = y
76.     if comb[j][z[j]] is z[j] then z[j] = z[j] + s
77.     else if comb[j][z[j] − 1] is not z[j] − 1 then z[j] = z[j] − s
78. }

/* Returns whether comb[i] is increasing. */
79. function inc(i) {
80.     return comb[j][i + 1] mod 2 is d[j] > 0
81. }

/* Returns whether comb[i] and [i+1] are adjacent. */
82. function adj(i) {
83.     return comb[j][i] + 1 is comb[j][i + 1]
84. }
```

A sample output for MULTPERM is shown in Figure 5.1, for the multiset  $\{1, 1, 2, 2, 3\}$ . In each column, the 1s travel through all possible positions. Note that it is coincidental that the 1s finish as a group at either side: Chase’s algorithm dictates that more than one element travelling through five possible positions will finish with exactly two elements maximal, which just happens to be the number of 1s in this example. From the bottom of each column to the top of the next, the 2s travel through the subpermutation including the 3 by the same algorithm. A complete C program for MULTPERM is given in Appendix H.

The majority of the variables in MULTPERM are inherited from its component algorithms and used in their usual ways. One small modification is

that Williamson’s algorithm has been modified to select 1 first, rather than  $n$ , meaning that the 1s move first in MULTPERM. A more significant modification is that, since each of the  $k$  groups of distinct elements is represented by its own combination, array *comb* and variable  $z$  are each extended by  $k$  elements in one dimension. Thus, instead of operating on *comb*[ $i$ ] and  $z$ , Chase’s algorithm operates on *comb*[ $j$ ][ $i$ ] and  $z[j]$ , where  $j$  is the currently selected group of similar elements. Each *comb*[ $i$ ] is of length  $m_i$ , meaning the entire array is of length  $n$ , thus achieving linear space.

A few new variables were introduced, starting with array *perm*, which holds the permutation. A new array, *o*, was introduced to maintain the offsets of each group of similar elements within the permutation. This is used to translate changes from Chase’s algorithm on some *comb*[ $i$ ] to *perm*. Each group of elements affects the offsets of every subsequent group of elements, and time-stealing (Section 3.2) is used to update these  $O(k)$  offsets. For example, if the first multiset is  $\{1, 1, 2, 3, 4\}$ , then when the 1s finish moving the permutation will be  $\{2, 3, 4, 1, 1\}$ , so the 2 and the 3 should have their offsets reduced by 2. In turn, the movement of the 2 will affect the offset of the 3. Note that the *ns*, or in this case the 4s, are never selected for movement and thus do not require an offset. Array *a* is used to keep track of which subsequent offsets are due to be updated for some  $j$ . Thus, *a*[ $j$ ] begins at  $j + 1$  and counts up to  $k$ , at a rate of one per iteration of Chase’s algorithm. For the above example, during the movement of the 1s, *a*[1] will count up from 2 to 4, updating so *o*[2] and *o*[3], then terminating. Variable *a*[ $j$ ] is reinitialised on line 16, the only reinitialisation required for Chase’s algorithm.

Two small modifications have been made to enable Chase’s algorithm to reverse, as discussed in Section 5.2. The extra terminating condition has been added, such that it will halt after generating what is normally the initial object, on line 12. Also, function *inc* has been tied to the direction variable of Williamson’s algorithm, as shown on line 80. Thus, if Williamson’s algorithm says element  $j$  is decreasing, then Chase’s algorithm will run backwards for the combination of  $j$ s. Since inverting *d*[ $j$ ] is already part of Williamson’s algorithm, no extra reinitialisation code is needed.

Thus far this chapter has described the design and development of MULT-

PERM. The next section compares MULTPERM and Korsh & LaFollette’s (2004) recently published algorithm, which was the first loopless algorithm for multiset permutations to achieve the milestone of linear space using only arrays.

#### **5.4 Evaluation vs. Korsh & LaFollette’s Algorithm**

This section compares MULTPERM with the loopless algorithm for multiset permutations recently published by Korsh & LaFollette, here labelled “KL04”. The algorithms are compared in terms of running time, space required, and, informally, code size. MULTPERM proves to be faster, smaller and simpler than Korsh & LaFollette’s algorithm, reasons for which are discussed with regard to the fusing design process.

The running times of MULTPERM and KL04 for generating large listings of permutations were experimentally measured over two different types of multisets. The multisets had multiplicities  $\{3, 3, 3, 3, 3\}$  and  $\{2, 3, 5, 2, 3\}$ , and were labelled “uniform” and “varied” respectively. Thus, the former was the multiset comprising three 1s, three 2s, and so on, while the latter was the multiset comprising two 1s, three 2s, five 3s, and so on. They required the generation of over 168 million and over 75 million permutations respectively.

Both algorithms were implemented in C using similar programming style, given in Appendices H and I. As is conventional, output statements were removed, and a counter incremented instead. This removes the very high overhead of I/O system calls, allowing for more accurate measurement of the algorithms’ generating times. The final value for the counter is output so that the number of permutations generated can be verified. Mean times were calculated from eight iterations of each algorithm with each multiset. The Python script for running this experiment and the raw data generated are given in Appendices J and K respectively. The test computer was an AMD Athlon processor running at 1.16 GHz, with 256MB of RAM, and running the Windows XP operating system.

As can be seen from Table 5.1, MULTPERM ran 44–46% faster than KL04 across both multisets. MULTPERM generated the 168 million permutations of the uniform-multiplicities multiset in an average of 27.8s ( $\sigma = 0.02$ )

Uniform		
	KL04	MULTPERM
<b>Permutations</b>	168,168,000	168,168,000
<b>Mean Time (s)</b>	49.4	27.8

Varied		
	KL04	MULTPERM
<b>Permutations</b>	75,675,600	75,675,600
<b>Mean Time (s)</b>	22.4	12.1

**Table 5.1** Experimental times showing that MULTPERM generates multiset permutations 44–46% faster than KL04. Testing was over multisets with uniform and varied multiplicities.

to KL04’s 22.4s ( $\sigma = 0.04$ ), and the 75 million permutations of the varied multiset in 12.1s ( $\sigma = 0.03$ ) to KL04’s 22.4s ( $\sigma = 0.06$ ). Thus, MULTPERM runs significantly faster.

An estimate of space requirements can be arrived at by counting the number and sizes of arrays used by each algorithm. As can be seen from Appendices H and I, MULTPERM has two arrays of size  $n$  and eight of size  $k$ , while KL04 has eight arrays of size  $n$  and eleven of size  $k$ . Variables  $n$  and  $k$  are the size of the multiset and its number of distinct elements respectively. Thus, as  $k$  tends towards  $n$ , MULTPERM’s space requirement tends towards  $10n$  and KL04’s towards  $19n$ . Conversely, as  $k$  tends towards 1, the spaces used tend towards  $2n$  and  $8n$  respectively. Thus, depending on the input multiset, MULTPERM requires between a half and a quarter of the space required by KL04.

Finally, since the two appendices have been coded in similar style, some estimate of relative program complexity can be made by counting lines of code. The program for MULTPERM requires 136 lines, while that for KL04 requires 311. It seems fair to conclude that MULTPERM is the simpler algorithm. Thus, MULTPERM is faster, smaller and simpler than KL04.

This chapter applied the general framework for fusing loopless algorithms to develop a new loopless algorithm for generating multiset permutations. The new algorithm proved faster than that recently published by Korsh &

LaFollette, which was developed using a similar approach. The extra speed and simplicity of MULTPERM over Korsh & LaFollette's algorithm is due to the use here of Chase's combinations algorithm, which the design process of Section 5.1 permitted, but which that of Korsh & LaFollette precluded. As a component algorithm in fusing, Chase's algorithm easy reversibility and constant-distance transpositions made it more advantageous than that of Eades & McKay, which was used by Korsh & LaFollette. This suggests certain properties of component algorithms can significantly affect the efficiency of the final fused algorithms.

# Chapter VI

## Discussion

This chapter summarises the results of this thesis, listing its main achievements and addressing the initial questions it set out to answer. It then looks at possible directions for future work based on this research.

### 6.1 Results

The main achievements of this thesis were:

- Developing a general framework for fusing loopless algorithms, including two distinct structures and techniques for reinitialisation that are applicable to both (Chapter 3);
- Successfully applying this framework to develop two new loopless algorithms, for mixed parenthesis strings (Chapter 4) and multiset permutations (Chapter 5) respectively;
- For parenthesis strings, developing a novel method for finding the positions of right parentheses in  $O(1)$  time (Section 4.2);
- For multiset permutations, using Chase’s algorithm as a component to produce an algorithm faster, smaller and simpler than the previous best, that of Korsh & LaFollette (Sections 5.3 and 5.4);
- Publishing these results at a refereed conference, the twelfth *Computing: The Australasian Theory Symposium* (Takaoka & Violich 2006).

Regarding the first question this thesis set out to answer, whether the approach of fusing loopless algorithms could be generalised, the answer is



undoubtedly positive. Chapter 3 developed a framework for fusing loopless algorithms, offering two different structures. The first permits one loopless algorithm to be fully nested inside the other. The second structure shows how a loopless algorithm can be fused within Williamson’s algorithm. Section 3.2 developed concepts about reinitialising loopless algorithms that are applicable to both structures. Reinitialisation is a critical step, because fusing loopless algorithms implies that one must be run repeatedly, something loopless algorithms are generally not designed for. Thus, these techniques have now been presented as a single coherent framework.

The second question, whether the approach is a generally useful tool, is perhaps best answered with cautious optimism. Each of the above structures was successfully applied to develop a new loopless algorithm. Chapter 4 applied the nesting structure to develop a loopless algorithm for generating mixed parenthesis strings. MIXPAR is the first loopless algorithm to generate these objects, and achieved  $O(n)$  space complexity, which is the minimum possible. Chapter 5 applied the Williamson’s algorithm structure to develop a loopless algorithm for generating multiset permutations. Although a similar approach has been tried in the past, MULTPERM almost became the first such algorithm achieved linear space using only arrays, beaten to this milestone during this research by the recent publication of Korsh & LaFollette. Section 5.4 showed MULTPERM to be much faster than that algorithm, and as can be seen by comparing the code in Appendices H and I, MULTPERM is also more concise. Thus, the approach proved useful for tackling these two combinatorial generation problems, but it is not yet clear how much more widely it can be applied.

## **6.2 Future Work**

One specific area for future work is to improve on the new algorithm for multiset permutations. While Chase’s algorithm for combinations proved advantageous, there may yet be more advantageous algorithms that have not yet been tried. Ruskey’s (1993) algorithm for combinations, for example, achieves constant-distance transpositions using a simpler construction than Chase’s. This algorithm is recursive, but if it can be implemented looplessly

it could make for an even more efficient loopless algorithm for multiset permutations. Another, more recent algorithm by Ruskey & Williams (2005) looplessly generates combinations in surprisingly little code by prefix rotation. Either or both of these algorithms might contribute to a more efficient version of MULTPERM.

A more general area for future work is to investigate how widely applicable the approach of fusing loopless algorithms is. These two applications, though quite successful, do not provide a large enough sample from which to draw such a conclusion. One particular problem marked for future attention is multiset combinations. There may well be other problems where the approach of fusing loopless algorithms for combinatorial generation is useful.

Another possible avenue for exploration is whether the process of fusing algorithms, loopless or otherwise, can be automated. For example, if the algorithms' inputs and outputs were suitably specified, could a programming language automatically fuse them to combine their effects? This research would branch away from algorithms and more towards programming languages themselves.

In conclusion, this thesis was successful in its aims to generalise and apply a framework for fusing loopless algorithms. The two new algorithms presented suggest that the method can be useful, although further work is needed to gauge how widely applicable the approach is. As discussed in the preceding three paragraphs, there is plenty of scope for further research to build on these results.

## References

- Canfield, E. R. & Williamson, S. G. (1995), ‘A loop-free algorithm for generating linear extensions of a poset’, *Order* **12**.
- Chase, P. J. (1989), ‘Combination generation and graylex ordering’, *Congressus Numerantium* **69**, 215–242.
- Eades, P. & McKay, B. (1984), ‘An algorithm for generating subsets of fixed size with a strong minimal change property’, *Inform. Process. Lett.* **19**, 131–133.
- Ehrlich, G. (1973), ‘Loopless algorithms for generating permutations, combinations, and other combinatorial configurations’, *J. ACM* **20**(3), 500–513.
- Gray, F. (1953), ‘Pulse code communication’, U.S. patent no. 2,632,058.
- Johnson, S. M. (1963), ‘The generation of permutations by adjacent transpositions’, *Math. Comp.* **17**, 282–285.
- Korsh, J. F. & LaFollette, P. S. (2004), ‘Loopless array generation of multiset permutations’, *The Computer Journal* **47**(5), 612–621.
- Korsh, J. & Lipschutz, S. (1997), ‘Generating multiset permutations in constant time’, *J. Alg.* **25**(2), 321–335.
- Lehmer, D. H. (1964), The machine tools of combinatorics, in E. F. Beckenbach, ed., ‘Applied Combinatorial Mathematics’, Wiley, pp. 5–31.
- Nijenhuis, A. & Wilf, H. S. (1975), *Combinatorial Algorithms*, Academic Press.

- Pruesse, G. & Ruskey, F. (1994), ‘Generating linear extensions fast’, *SIAM Journal on Computing* **23**(2), 373–386.
- Reingold, E. M., Nievergelt, J. & Deo, N. (1977), *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall.
- Ruskey, F. (1993), Simple combinatorial gray codes constructed by reversing sublists, in ‘4th ISAAC (International Symposium on Algorithms and Computation)’, Vol. 762 of *Lecture Notes in Computer Science*, pp. 201–208.
- Ruskey, F. & Williams, A. (2005), Generating combinations by prefix shifts., in ‘11th COCOON (Computing and Combinatorics International Conference)’, Vol. 3595 of *Lecture Notes in Computer Science*, pp. 570–576.
- Savage, C. (1997), ‘A survey of combinatorial gray codes’, *SIAM Review* **39**(4), 605–629.
- Takaoka, T. (1999), An  $O(1)$  time algorithm for generating multiset permutations, in ‘10th ISAAC (International Symposium on Algorithms and Computation)’, Vol. 1741 of *Lecture Notes in Computer Science*, pp. 237–246.
- Takaoka, T. & Violich, S. (2006), Combinatorial Generation by Fusing Loopless Algorithms, in ‘12th CATS (Computing: The Australasian Theory Symposium)’, Vol. 51 of *Conferences in Research and Practice in Information Technology*, pp. 237–246.
- Trotter, H. F. (1962), ‘Perm (algorithm 115)’, *Commun. ACM* **5**(8), 434–435.
- Vajnovszki, V. (2003), ‘A loopless algorithm for generating the permutations of a multiset’, *Theor. Comput. Sci.* **307**(2), 415–431.
- Wilf, H. S. (1989), *Combinatorial Algorithms: An Update*, SIAM.
- Williamson, S. G. (1985), *Combinatorics for Computer Science*, Computer Science Press.

Xiang, L. & Ushijima, K. (2001), ‘On  $O(1)$  time algorithms for combinatorial generation’, *Comput. J.* **44**(4), 292–302.

## Appendix A

### grayrec.c

```
/* A recursive algorithm for generating binary Gray codes. */

#include <stdio.h>
#include <stdlib.h>

int n, *bin, i, cnt = 0;

void print(), next(int j);

/* Main. */
void main() {
    printf("n: ");
    scanf("%d", &n);
    bin = (int *)calloc(n+1, sizeof(int));
    for (i=1; i<=n; i++) bin[i] = 0;
    print();
    next(1);
}

/* Recursively generates all Gray codes rooted at bin[j]. */
void next(int j) {
    if (j == n) {
        bin[j] = 1-bin[j];
        print();
    } else {
        next(j+1);
        bin[j] = 1-bin[j];
        print();
        next(j+1);
    }
}
```

```
/* Prints bin. */  
void print() {  
    cnt++;  
    printf("%d. ", cnt);  
    for (i=1; i<=n; i++) printf("%d ", bin[i]);  
    printf("\n");  
}
```

## Appendix B

### graylpl.c

```
/* A loopless algorithm for generating binary Gray codes. */

#include <stdio.h>
#include <stdlib.h>

int n, *bin, j, *e, i, cnt = 0;

void print(), next();

/* Main. */
void main() {
    printf("n: ");
    scanf("%d", &n);
    bin = (int *)calloc(n+1, sizeof(int));
    e = (int *)calloc(n+1, sizeof(int));
    for (i=1; i<=n; i++) bin[i] = 0;
    for (i=0; i<=n; i++) e[i] = i;
    j = n;
    while (j != 0) {
        print();
        next();
    }
    print();
}

/* Looplessly generates the next binary string. */
void next() {
    e[n] = n;
    bin[j] = 1-bin[j];
    e[j] = e[j-1];
    e[j-1] = j-1;
```



```
    j = e[n];  
}  
  
/* Prints bin. */  
void print() {  
    cnt++;  
    printf("%d. ", cnt);  
    for (i=1; i<=n; i++) printf("%d ", bin[i]);  
    printf("\n");  
}
```

## Appendix C

### will.c

```
/* Williamson's (1985) loopless algorithm for generating variations in
 * Gray code order. */

#include <stdio.h>
#include <stdlib.h>

int n, *r, *var, j, *e, *d, i, cnt = 0;

void init(), next(), print();

/* Main. */
void main() {
    init();
    print();
    while (j != 0) {
        next();
        print();
    }
}

/* Initialisation. */
void init() {
    printf("n: ");
    scanf("%d", &n);
    r = (int *)calloc(n+1, sizeof(int));
    var = (int *)calloc(n+1, sizeof(int));
    e = (int *)calloc(n+1, sizeof(int));
    d = (int *)calloc(n+1, sizeof(int));
    for (i=1; i<=n; i++) {
        printf("r[%d]: ", i);
```

```

        scanf("%d", &r[i]);
    }
    for (i=1; i<=n; i++) var[i] = 0;
    for (i=0; i<=n; i++) e[i] = i;
    j = n;
    for (i=1; i<=n; i++) d[i] = 1;

}

/* Looplessly generates the next variation. */
void next() {
    e[n] = n;
    var[j] = var[j]+d[j];
    if (var[j] == r[j]-1 || var[j] == 0) {
        e[j] = e[j-1];
        e[j-1] = j-1;
        d[j] = -d[j];
    }
    j = e[n];
}

/* Prints var. */
void print() {
    cnt++;
    printf("%d. ", cnt);
    for (i=1; i<=n; i++) printf("%d ", var[i]);
    printf("\n");
}

```

## Appendix D

### xiang.c

```
/* Xiang & Ushijima's (2001) loopless algorithm for generating
 * parenthesis strings */

#include <stdio.h>
#include <stdlib.h>

char *par;
int n, *l, j, *e, *d, i, temp, cnt = 0;

void init(), print(), next();

/* Main. */
void main() {
    init();
    print();
    while (j != 1) {
        next();
        print();
    }
}

/* Initialisation. */
void init() {
    printf("n: ");
    scanf("%d", &n);
    par = (char *)calloc(2*n+1, sizeof(char));
    l = (int *)calloc( n+1, sizeof(int));
    e = (int *)calloc( n+1, sizeof(int));
    d = (int *)calloc( n+1, sizeof(int));
    for (i=1; i<=2*n; i+=2) par[i] = '(', par[i+1] = ')';
    for (i=1; i<=n; i++) l[i] = 2*i-1;
```

```

    for (i=1; i<=n; i++) d[i] = 1;
    for (i=1; i<=n; i++) e[i] = i;
    j = n;
}

/* Looplessly generates the next parenthesis string. */
void next() {
    e[n] = n;
    temp = l[j];
    if (d[j] == 1) {
        if (l[j] == 2*j-1) l[j] = l[j-1]+1;
        else l[j]++;
    } else {
        if (l[j] == l[j-1]+1) l[j] = 2*j-1;
        else l[j]--;
    }
    par[l[j]] = '(', par[temp] = ')';
    if (l[j] > 2*j-3) {
        d[j] = -d[j];
        e[j] = e[j-1];
        e[j-1] = j-1;
    }
    j = e[n];
}

/* Prints par. */
void print() {
    cnt++;
    printf("%d. ", cnt);
    for (i=1; i<=2*n; i++) printf("%c ", par[i]);
    printf("\n");
}

```

## Appendix E

### mixpar.c

```
/* Takaoka & Violich's (2005) loopless algorithm for generating mixed
 * parenthesis strings */

#include <stdio.h>
#include <stdlib.h>

char *par;
int n, *l, j, jj, *e, *ee, *d, *r, *s, t, i, temp, cnt = 0;

void init(), print(), nextGray(), reinit(), nextPar(), find();

/* Main. */
void main() {
    init();
    do {
        do {
            print();
            nextGray();
        } while (jj != 0);
        print();
        reinit();
        nextPar();
    } while (e[1] != 0);
}

/* Initialisation. */
void init() {
    printf("n: ");
    scanf("%d", &n);
    par = (char *)calloc(2*n+1, sizeof(char));
    l = (int *)calloc(n+1, sizeof(int));
```

```

    e  = (int *)calloc( n+1, sizeof(int));
    ee = (int *)calloc( n+1, sizeof(int));
    d  = (int *)calloc( n+1, sizeof(int));
    r  = (int *)calloc( n+1, sizeof(int));
    s  = (int *)calloc(2*n+2, sizeof(int));
    for (i=1; i<=2*n; i+=2) par[i] = '(', par[i+1] = ')';
    for (i=1; i<=n; i++) l[i] = 2*i-1;
    for (i=0; i<=n; i++) e[i] = i, ee[i] = i;
    for (i=1; i<=n; i++) d[i] = 1;
    for (i=1; i<=2*n+1; i++) s[i] = i;
    j = n, jj = n;
    t = n;
}

/* Looplessly generates the next string by Gray code. */
void nextGray() {
    ee[n] = n;
    if (jj == t) find();
    if (par[l[jj]] == '(') par[l[jj]] = '[', par[r[jj]] = ']';
    else par[l[jj]] = '(', par[r[jj]] = ')';
    ee[jj] = ee[jj-1];
    ee[jj-1] = jj-1;
    jj = ee[n];
}

/* Finds the right parenthesis in the jjth pair. */
void find() {
    if (par[l[1]] == '(') {
        r[jj] = s[l[jj]+1];
        if (jj != 1) {
            s[l[jj]] = s[r[jj]+1];
            t--;
        } else t = 2;
    } else {
        s[l[jj]] = l[jj];
        t++;
    }
}

/* Reinitialisation. */
void reinit() {

```

```

    par[l[1]] = '(', par[r[1]] = ')';
    jj = n;
    t = n;
}

/* Looplessly generates the next string by Xiang & Ushijima's algorithm. */
void nextPar() {
    e[n] = n;
    temp = l[j];
    if (d[j] == 1) {
        if (l[j] == 2*j-1) l[j] = l[j-1]+1;
        else l[j]++;
    } else {
        if (l[j] == l[j-1]+1) l[j] = 2*j-1;
        else l[j]--;
    }
    par[l[j]] = '(', par[temp] = ')';
    if (l[j] > 2*j-3) {
        d[j] = -d[j];
        e[j] = e[j-1];
        e[j-1] = j-1;
    }
    j = e[n];
}

/* Prints par. */
void print() {
    cnt++;
    printf("%d. ", cnt);
    for (i=1; i<=2*n; i++) printf("%c ", par[i]);
    printf("\n");
}

```



## Appendix F

### chase.c

```
/* Chase's (1989) loopless algorithm for generating combinations
 * by 1- or 2-apart transpositions. */

#include <stdio.h>
#include <stdlib.h>

int m, n, *c, z, x, y, i, j, cnt = 0;

void init(), next(), print();

/* Main. */
void main() {
    init();
    do {
        print();
        next();
    } while (y <= n);
}

/* Initialisation. */
void init() {
    printf("m: ");
    scanf("%d", &m);
    printf("n: ");
    scanf("%d", &n);
    c = (int *)calloc(m+2, sizeof(int));
    for (i=1; i<=m; i++) c[i] = i;
    c[m+1] = n+n+1;
    z = m+1;
}
```

```

/* Looplessly generates the next combination. Sets x to the outgoing
 * value and y to the incoming value. */
void next() {
    if (c[2]%2 == 0) {
        /* 50 */
        if (c[1] == 1) {
            /* 20 */
            if (c[z]%2 == 0) {
                /* 30 */
                if (c[z+1] == c[z]+1) {
                    /* 40 */
                    i = c[z+2]%2;
                    j = 3*i-1;
                    x = c[z+1]-i;
                    y = c[z]+j;
                    c[z+1] = x+j;
                    c[z] = y-i;
                    if (y == z) z = x;
                } else {
                    /* 30 cont */
                    j = c[z+1]%2;
                    j = j+j-1;
                    x = c[z];
                    y = x+j;
                    c[z] = y;
                    if (y == z) z = x;
                }
            } else {
                /* 20 cont */
                y = z;
                x = z+z%2-2;
                c[z-1] = z;
                z = x;
                c[x] = x+1;
            }
        } else {
            /* 50 cont */
            x = c[1];
            z = 2;
            y = x-1;
            c[1] = y;

```

```

        }
    } else if (c[2] == c[1]+1) {
        /* 10 */
        i = c[3]%2;
        z = 3;
        c[1] = c[1]+i+i-1;
        x = c[2]-i;
        c[2] = c[1]+1;
        y = c[1]+i;
    } else {
        x = c[1];
        y = c[1]+1;
        c[1] = y;
    }
}

/* Prints the combination. */
void print() {
    cnt++;
    printf("%d. ", cnt);
    for (i=1; i<=m; i++) printf("%d ", c[i]);
    printf("\n");
}

```

## Appendix G

### chasemod.c

```
/* A modified version of Chase's (1989) loopless algorithm for
 * generating combinations by 0(1)-distance transpositions. This
 * version is more readable and can be modified to run backwards. */

#include <stdio.h>
#include <stdlib.h>

int n, r, *comb, z, x, y, b, i, cnt = 0;

void init(), next(), move(int,int,int), print();
int inc(int), adj(int);

/* Main. */
void main() {
    init();
    print();
    while (comb[n-b] != n-b || comb[n-b+1] != r-b+1) {
        next();
        print();
    }
}

/* Initialisation. */
void init() {
    printf("n: ");
    scanf("%d", &n);
    printf("r: ");
    scanf("%d", &r);
    comb = (int *)calloc(n+2, sizeof(int));
    for (i=1; i<=n; i++) comb[i] = i;
    comb[n+1] = 2*r+1;
```

```

    z = n+1;
    if (r%2 == 0 || n == 1) b = 1; else b = 2;
}

/* Looplessly generates the next combination by selecting one
 * of ten movements. */
void next() {
    if (z == 1) {
        if (inc(1)) {
            if (adj(1)) {
                if (inc(2)) move(1, 1, 2);
                else move(2, -1, 2);
            } else move(1, 1, 1);
        } else move(1, -1, 1);
    } else {
        if (inc(z-1)) {
            if (z>2 && inc(z-2)) move(z-2, 1, 2);
            else move(z-1, 1, 1);
        } else {
            if (!adj(z)) {
                if (inc(z)) move(z, 1, 1);
                else move(z, -1, 1);
            } else {
                if (inc(z+1)) move(z, 1, 2);
                else move(z+1, -1, 2);
            }
        }
    }
}

/* Moves an element for a given position, direction and span. */
void move(int p, int d, int s) {
    x = comb[p];
    y = x+s*d;
    comb[p] = x+d;
    comb[p+d*(s-1)] = y;
    if (comb[z] == z) z = z+s;
    else if (comb[z-1] != z-1) z = z-s;
}

/* Returns whether comb[i] is increasing. */

```

```

int inc(int i) {
    return comb[i+1]%2;
}

/* Returns whether comb[i] and [i+1] are adjacent. */
int adj(int i) {
    return comb[i]+1 == comb[i+1];
}

/* Prints comb. */
void print() {
    cnt++;
    printf("%d. ", cnt);
    for (i=1; i<=n; i++) printf("%d ", comb[i]);
    printf("\n");
}

```

## Appendix H

### multperm.c

```
/* Takaoka & Violich's (2005) loopless algorithm for generating
 * multiset permutations in linear space using only arrays. */

#include <stdio.h>
#include <stdlib.h>

int k, *m, n, *perm, **comb, *z, x, y, j, *e, *d, *o, *r, *a, *b, i, cnt = 0;

int adj(int), inc(int);
void init(), nextWill(), nextChase(), move(int,int,int), print();

/* Main. */
void main() {
    init();
    print();
    while (j != k) {
        nextWill();
        print();
    }
    /* Uncomment next line when timing. */
    /* printf("%d\n", cnt); */
}

/* Initialisation. */
void init() {
    fprintf(stderr, "k: ");
    scanf("%d", &k);
    n = 0;
    m = (int *)calloc(k+1, sizeof(int));
    for (i=1; i<=k; i++) {
        fprintf(stderr, "m[%d]: ", i);
```

```

        scanf("%d", &m[i]);
        n = n+m[i];
    }
    perm = (int *)calloc(n+1, sizeof(int));
    comb = (int **)calloc(k+1, sizeof(int *));
    for (i=1; i<=k; i++) comb[i] = (int *)calloc(m[i]+2, sizeof(int));
    z = (int *)calloc(k+1, sizeof(int));
    e = (int *)calloc(k+2, sizeof(int));
    d = (int *)calloc(k+1, sizeof(int));
    o = (int *)calloc(k+1, sizeof(int));
    r = (int *)calloc(k+1, sizeof(int));
    a = (int *)calloc(k, sizeof(int));
    b = (int *)calloc(k, sizeof(int));
    o[1] = 0;
    for (i=2; i<=k; i++) o[i] = o[i-1]+m[i-1];
    r[k] = m[k];
    for (i=k-1; i>=1; i--) r[i] = r[i+1]+m[i];
    for (i=1; i<=k; i++) {
        for (j=1; j<=m[i]; j++) perm[j+o[i]] = i;
    }
    for (i=1; i<=k; i++) d[i] = 1;
    for (i=0; i<=k+1; i++) e[i] = i;
    for (i=1; i<=k; i++) {
        for (j=1; j<=m[i]; j++) comb[i][j] = j;
        comb[i][m[i]+1] = 2*r[i]+1;
        z[i] = m[i]+1;
    }
    for (i=1; i<=k-1; i++) {
        a[i] = i+1;
        b[i] = 1+(m[i]>1 && r[i]%2);
    }
    j = 1;
}

/* Looplessly selects the next j by Williamson's algorithm. */
void nextWill() {
    e[1] = 1;
    nextChase();
    if (comb[j][m[j]-b[j]+1] == r[j]-b[j]+1
    && comb[j][m[j]-b[j]] == m[j]-b[j]
    || comb[j][m[j]] == m[j]) {

```



```

        e[j] = e[j+1];
        e[j+1] = j+1;
        d[j] = -d[j];
        a[j] = j+1;
    }
    j = e[1];
}

/* Looplessly generates the next permutation by Chase's algorithm. */
void nextChase() {
    if (z[j] == 1) {
        if (inc(1)) {
            if (adj(1)) {
                if (inc(2)) move(1, 1, 2);
                else move(2, -1, 2);
            } else move(1, 1, 1);
        } else move(1, -1, 1);
    } else {
        if (inc(z[j]-1)) {
            if (z[j]>2 && inc(z[j]-2)) move(z[j]-2, 1, 2);
            else move(z[j]-1, 1, 1);
        } else {
            if (!adj(z[j])) {
                if (inc(z[j])) move(z[j], 1, 1);
                else move(z[j], -1, 1);
            } else {
                if (inc(z[j]+1)) move(z[j], 1, 2);
                else move(z[j]+1, -1, 2);
            }
        }
    }
    } /* Values for x and y were produced by move. */
    perm[x+o[j]] = perm[y+o[j]], perm[y+o[j]] = j;
    if (a[j]<k) {
        o[a[j]] = o[a[j]]-b[j]*d[j];
        a[j] = a[j]+1;
    }
}

/* Moves an element for a given position, direction and span. */
/* Produces new values for global variables x and y. */
void move(int p, int d, int s) {

```

```

    x = comb[j][p];
    y = x+s*d;
    comb[j][p] = x+d;
    comb[j][p+d*(s-1)] = y;
    if (comb[j][z[j]] == z[j]) z[j] = z[j]+s;
    else if (comb[j][z[j]-1] != z[j]-1) z[j] = z[j]-s;
}

/* Returns whether comb[i] is increasing. */
int inc(int i) {
    return comb[j][i+1]%2 == d[j]>0;
}

/* Returns whether comb[i] and [i+1] are adjacent. */
int adj(int i) {
    return comb[j][i]+1 == comb[j][i+1];
}

/* Prints perm. */
void print() {
    cnt++;
    /* Comment out next three lines when timing. */
    printf("%d. ", cnt);
    for (i=1; i<=n; i++) printf("%d ", perm[i]);
    printf("\n");
}

```

# Appendix I

## korsh.c

```
/* Korsh and LaFollette's (2004) loopless algorithm for generating
 * multiset permutations in linear space using only arrays. */

#include <stdio.h>
#include <stdlib.h>

int i, j, J, dir, m, n, k, t, M0, ctr, LEFT=1, top, L, end, *delta,
    *jj, *tt, *MM0, *cctr, *E, *d, *LL, *eend, *K, *N, *p, *C, *JN, *NUM, *last,
    *Last, *f, *F, **CC, **llast, **LLast, **ff, **FF, **JJN, **NNUM, cnt = 0;

void init(), print(), next();

int main() {
    init();
    print();
    while (J != 1) {
        next();
        print();
    }
    /* Uncomment out next line when timing. */
    /* printf("%d\n", cnt); */
}

void init() {
    fprintf(stderr, "m: ");
    scanf("%d", &m);
    N=(int *)calloc(m+1, sizeof(int));
    N[0]=0;
    K=(int *)calloc(m+1, sizeof(int));
    MM0=(int *)calloc(m+1, sizeof(int));
    jj=(int *)calloc(m+1, sizeof(int));
```

```

tt=(int *)calloc(m+1, sizeof(int));
d=(int *)calloc(m+1, sizeof(int));
delta=(int *)calloc(m+1, sizeof(int));
ctr=(int *)calloc(m+1, sizeof(int));
LL=(int *)calloc(m+1, sizeof(int));
eend=(int *)calloc(m+1, sizeof(int));
E=(int *)calloc(m+2, sizeof(int));
for (i=1; i<=m; i++) {
    fprintf(stderr, "K[%d]: ", i);
    scanf("%d", &K[i]);
    N[i]=N[i-1]+K[i];
    MMO[i]=K[i];
}
CC=(int **)calloc(m+1, sizeof(int *));
ff=(int **)calloc(m+1, sizeof(int *));
FF=(int **)calloc(m+1, sizeof(int *));
llast=(int **)calloc(m+1, sizeof(int *));
LLast=(int **)calloc(m+1, sizeof(int *));
JJN=(int **)calloc(m+1, sizeof(int *));
NNUM=(int **)calloc(m+1, sizeof(int *));
p=(int *)calloc(N[m]+1, sizeof(int));
for (i=1; i<=m; i++) {
    for (j=N[i-1]+1; j<=N[i]; j++) p[j]=i;
}
for (i=1; i<=m; i++) {
    int KL=K[i]+2;
    CC[i]=(int *)calloc(KL, sizeof(int));
    JJN[i]=(int *)calloc(KL, sizeof(int));
    LLast[i]=(int *)calloc(KL, sizeof(int));
    llast[i]=(int *)calloc(KL, sizeof(int));
    NNUM[i]=(int *)calloc(KL, sizeof(int));
    ff[i]=(int *)calloc(KL, sizeof(int));
    FF[i]=(int *)calloc(KL, sizeof(int));
    d[i]=!LEFT;
    E[i]=i-1;
    delta[i]=0;
    int s=N[i-1];
    if (s/2*2!=s) LLast[i][1]=MMO[i]+s;
    else LLast[i][1]=s+1;
    ctr[i]=s-1;
    jj[i]=N[i-1];
}

```

```

        tt[i]=N[i-1]+1;
        LL[i]=eend[i]=1;
        llast[i][1]=MMO[i]+1;
        JJN[i][0]=0;
        JJN[i][1]=NNUM[i][1]=CC[i][1]=N[i-1];
        ff[i][1]=FF[i][1]=0;
    }
    J=m;
    k=N[J-1];
    n=N[J];
    dir=d[J];
    MO=MMO[J];
    j=jj[J];
    t=tt[J];
    C=CC[J];
    L=LL[J];
    end=eend[J];
    last=llast[J];
    Last=LLast[J];
    f=ff[J];
    F=FF[J];
    JN=JJN[J];
    NUM=NNUM[J];
    ctr=cctr[J];
}

/* Prints the permutation. */
void print() {
    cnt = cnt+1;
    /* Comment out next three lines when timing. */
    printf("%d. ", cnt);
    for (i=1; i<=N[m]; i++) printf("%d ", p[i]);
    printf("\n");
}

void next() {
    // Use... (see comment for any hope of explaining this)
    int S=C[L]-(JN[L]-j);
    int Mj=MO+j;
    if (((dir==LEFT)&&(j/2*2==j))||((dir!=LEFT)&&(j/2*2!=j))) {
        if (S==Mj) {

```

```

C[L-1]++;
NUM[L-1]++;
JN[L-1]++;
Last[L-1]=last[L];
last[L]=C[L-1]+1;
if (S-1!=C[L-1]+1) {
    if (NUM[L]==1) {
        last[L]=Last[L+1];
        last[L]=C[L]+1;
        end=L-1;
    } else if (NUM[L]==2) Last[L]=last[L];
    NUM[L]--;
} else {
    if (NUM[L]>1) NUM[L]--;
    else end=L-1;
    if (NUM[L]==1) Last[L]=last[L];
}
L--;
} else {
    if (NUM[L]>1) {
        JN[L]--;
        NUM[L]--;
        C[L]--;
        last[L+1]=Last[L];
        if (L<end) NUM[L+1]=NUM[L+1]+1;
        else {
            Last[L+1]=Last[L];
            JN[L+1]=k;
            NUM[L+1]=1;
            C[L+1]=C[L]+2;
            end=L+1;
        }
        Last[L]=S; L++;
    } else {
        if (L<end) {
            if (C[L]+1==C[L+1]-(JN[L+1]-j)) {
                JN[L]=JN[L+1];
                NUM[L]=NUM[L+1]+1;
                Last[L]=Last[L+1];
                C[L]=C[L+1];
                end=L;
            }
        }
    }
}

```

```

        }
        } else C[L]++;
    }
}
} else {
    if ((NUM[L]>1)&&(JN[L]==j)) {
        JN[L]--;
        NUM[L]--;
        C[L]--;
        last[L+1]=Last[L];
        if (L<end) {
            NUM[L+1]++;
            if (NUM[L]==1) Last[L]=last[L];
            else Last[L]=Mj-1;
        } else {
            C[L+1]=Mj;
            JN[L+1]=k;
            NUM[L+1]=1;
            Last[L+1]=Last[L];
            if (NUM[L]>1) Last[L]=Mj-1;
            else Last[L]=last[L];
        }
        end=L+1;
        L++;
    } else {
        if (L==end) {
            if ((S-1!=C[L-1]+1)|| (L==1)) {
                JN[L+1]=JN[L];
                NUM[L+1]=NUM[L]-1;
                C[L+1]=C[L];
                Last[L+1]=Last[L];
                Last[L]=last[L];
                last[L+1]=S;
                if (NUM[L+1]==1) Last[L+1]=S;
                JN[L]=j;
                NUM[L]=1;
                C[L]=S-1;
                last[L]=Last[L];
                if (NUM[L+1]>0) end=L+1;
                else end=L;
            } else {

```

```

        C[L-1]++;
        NUM[L-1]++;
        JN[L-1]++;
        Last[L-1]=last[L];
        if (NUM[L]>1) {
            last[L]=S;
            NUM[L]--;
            if (NUM[L]==1) Last[L]=last[L];
        }
        else end=L-1;
        L--;
    }
}

}

int newS=C[L], LAST=Last[L];
if (JN[L]!=j) {
    newS=C[L]-(JN[L]-j);
    LAST=last[L];
}

// transpose them
int I1=S+delta[J], I2=newS+delta[J], temp=p[I1];
p[I1]=p[I2];
p[I2]=temp;
top=k;
if (JN[L]==j) {
    if (newS==LAST) {
        Last[L]=S;
        if (NUM[L]==1) last[L]=S;
        F[L]=1;
        if (NUM[L]==1) {
            f[L]=1;
            if (L==end) top=j-1;
        }
        else if (f[L]) {
            if (L==end) top=JN[L-1];
        } else if (L==end) top=JN[L-1]+1;
    } else {
        F[L]=0;
        if (NUM[L]==1) f[L]=0;
    }
}

```



```

} else {
    if (newS==LAST) {
        last[L]=S;
        f[L]=1;
        if (L==end) top=j-1;
    } else f[L]=0;
}
if ((j<k)&&(newS==Mj)) {
    t=j;
    if (f[L]) j--;
}
else {
    if (t==j) t++;
    if (t<top) j=t;
    else j=top;
}
if (j<=JN[L]-NUM[L]) L--;
else if (j>JN[L]) L++;
if ((ctr!=0)&&(dir!=LEFT)) {
    if (JN[1]==ctr) {
        if (ctr/2*2!=ctr) Last[1]=M0+ctr;
        else Last[1]=ctr+1;
        F[L]=0;
        ctr--;
    }
}
E[m+1]=m;
if (j==0) {
    d[J]!=d[J];
    E[J+1]=E[J];
    E[J]=J-1;
    LL[J]=eend[J]=1;
    f[1]=F[1]=0;
    JN[1]=NUM[1]=k;
    if (dir==LEFT) {
        delta[J-1]=delta[J-1]-K[J];
        jj[J]=k;
        tt[J]=k+1;
        last[1]=M0+1;
        C[1]=k;
        cctr[J]=k-1;
    }
}

```

```

        if (k/2*2!=k) Last[1]=M0+k;
        else Last[1]=k+1;
    } else {
        delta[J-1]=delta[J-1]+K[J];
        jj[J]=1;
        tt[J]=2;
        last[1]=1;
        C[1]=n;
    }
} else {
    jj[J]=j;
    tt[J]=t;
    LL[J]=L;
    eend[J]=end;
    cctr[J]=ctr;
}
J=E[m+1];
k=N[J-1];
n=N[J];
dir=d[J];
M0=MM0[J];
j=jj[J];
t=tt[J];
C=CC[J];
NUM=NNUM[J];
JN=JJN[J];
f=ff[J];
F=FF[J];
last=llast[J];
Last=LLast[J];
L=LL[J];
end=eend[J];
ctr=cctr[J];
}

```

## Appendix J

### timer.py

```
# Python script for timing korsh and multperm algorithms over uniform
# and varied multperms.
# Usage: python timer.py > timedata.txt

import os
import random
import time

# Constants.
PROGS = ['korsh', 'multperm']
MULTS = [(3, 3, 3, 3, 3), (2, 3, 5, 2, 3)]
REPS = 8

# Variables.
progsMults = []
times = {}

# Initialise.
for prog in PROGS:
    for mult in MULTS:
        progsMults.append((prog, mult))
        times[(prog, mult)] = []

# Generate times.
for r in range(REPS):
    random.shuffle(progsMults) # Randomise order of trials.
    for prog, mult in progsMults:
        startClock = time.clock()
        progStdin, progStdout = os.popen2(prog)
        progStdin.write(str(len(mult))+'\n')
        for m in mult:
```

```

        progStdin.write(str(m)+'\n')
    progStdin.close()
    output = progStdout.read()
    progStdout.close()
    progTimeSecs = time.clock()-startClock
    numObj = long(output)
    times[(prog, mult)].append((numObj, progTimeSecs))

# Output times.
for mult in MULTS:
    print 'multiplicities\t' + str(mult)
    print 'objects gen. \t%.0f' % times[(PROGS[0], mult)][0][0]
    for prog in PROGS:
        print prog + ' t(s)',
        for r in range(REPS):
            print '\t%.2f' % times[(prog, mult)][r][1],
        print

```

## Appendix K

### timedata.txt

multiplicities	(3, 3, 3, 3, 3)							
objects gen.	168168000							
korsh t(s)	49.45	49.37	49.35	49.39	49.33	49.41	49.37	49.39
multperm t(s)	27.76	27.77	27.80	27.79	27.81	27.80	27.76	27.81
multiplicities	(2, 3, 5, 2, 3)							
objects gen.	75675600							
korsh t(s)	22.54	22.40	22.36	22.39	22.40	22.35	22.36	22.36
multperm t(s)	12.15	12.07	12.07	12.05	12.07	12.06	12.07	12.06